



Laravel 6 - Criando Aplicações Web

Nanderson Castro

Esse livro está à venda em <http://leanpub.com/laravel-6>

Essa versão foi publicada em 23/12/2019



* * * * *


Esse é um livro [Leanpub](http://leanpub.com). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](http://leanpub.com) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

* * * * *

© 2019 Nanderson Castro

Sumário

1. [Quem Somos](#)
 1. [Nanderson Castro](#)
 2. [Code Experts](#)
 3. [A loja virou a plataforma!](#)
2. [Para quem é este livro?](#)
3. [Laravel 6](#)
 1. [Obtendo o Laravel](#)
 2. [Iniciando Primeiro Projeto](#)
 3. [Conhecendo a estrutura do Laravel](#)
 4. [Laravel: Artisan CLI](#)
 5. [Executando a Aplicação](#)
 6. [Conclusões](#)
4. [Hello World com Laravel](#)
 1. [MVC](#)
 2. [Roteiro para nosso primeiro Hello World](#)
 3. [Iniciando Hello World](#)
 4. [Conclusões](#)
5. [Rotas & Controllers](#)
 1. [Rotas](#)
 2. [Controllers](#)
 3. [Conclusões](#)
6. [Formulários, Request & Response](#)
 1. [Iniciando pelo formulário](#)
 2. [Request, manipulando requisições](#)
 3. [Manipulando os dados da requisição](#)
 4. [Response, manipulando respostas](#)
 5. [Conclusões](#)
7. [Database: Migrations, Seeds & Factories](#)
 1. [Migrations](#)
 2. [Executando primeira migração](#)
 3. [Criando Nossas Migrações](#)
 4. [Relacionamentos via Migrations](#)
 5. [Seeds](#)
 6. [Factories](#)
 7. [Model Factories com Seeds](#)
 8. [Nossa primeira factory](#)
 9. [Migrations! Revertendo coisas!](#)
 10. [Conclusões](#)
8. [Eloquent, trabalhando com Models](#)
 1. [Os Models!](#)
 2. [Eloquent?](#)
 3. [Eloquent na prática](#)
 4. [Inserindo dados com Eloquent](#)
 5. [Mass Assignment](#)
 6. [Conclusões](#)
9. [Blade, Template Engine do Laravel](#)
 1. [Layouts](#)
 2. [Laços de Repetição & Condicionais](#)
 3. [Paginação na View](#)
 4. [Conclusões](#)
10. [Relacionamentos com Eloquent](#)
 1. [Relacionamento 1:N \(Um para Muitos e Inverso\)](#)
 2. [Inserindo Autor da Postagem](#)
 3. [ManyToMany com Eloquent: Categorias e Posts](#)
 4. [CRUD de Categorias](#)
 5. [Alterações em PostController](#)
 6. [Inserindo Muito para Muitos \(Post x Category\)](#)
 7. [Alterando Posts para Inserção N:N](#)
 8. [Conclusões](#)
11. [Autenticação](#)
 1. [Começando com a geração da autenticação](#)
 2. [As rotas de autenticação](#)

- 
3. [Middlewares](#)
 4. [Usando o middleware AUTH](#)
 5. [Blade, controles para autenticação](#)
 6. [Recuperando o usuário autenticado](#)
 7. [Relação 1:1 \(Autor e Perfil\)](#)
 8. [Criando Relação 1:1 nos Models](#)
 9. [Detalhes e melhorias](#)
 10. [Conclusões](#)
 12. [Upload de Arquivos](#)
 1. [Conhecendo Upload no Laravel](#)
 2. [Upload de Foto Perfil do Usuário](#)
 3. [Testando upload de foto do perfil](#)
 4. [Upload de Capa Postagem](#)
 5. [Conclusões](#)
 13. [Validação](#)
 1. [Form Request e Validações](#)
 2. [Montando Regras de Validação](#)
 3. [Usando o Form Request](#)
 4. [Exibindo validações nas Views Blade](#)
 5. [Exibindo validações nos formulário com Bootstrap](#)
 6. [Validação em Categorias](#)
 7. [Validações Perfil Usuário](#)
 8. [Mensagens de erro](#)
 9. [Conclusões](#)
 14. [Criando Front do nosso Blog](#)
 1. [Front do Blog](#)
 2. [Listagem de Posts e Single](#)
 3. [Criando Comentários](#)
 4. [Associando Comentários e Posts](#)
 5. [Salvando Comentários](#)
 6. [Postagens por Categorias](#)
 7. [Compartilhando Categorias entre as Views](#)
 8. [Exibindo categorias no menu](#)
 9. [Dinamizando Geração de Slugs Post & Category](#)
 10. [Conclusões](#)
 15. [Continue em contato conosco](#)

Quem Somos

Nanderson Castro

Iniciei na programação, contando o início da jornada e estudos, lá em 2007 mas meu contato com o primeiro computador foi nos anos 2000 por meio do computador de um tio meu com Windows 98 na época.

Meu principal intuito ao entrar no mundo da programação não era de nenhuma maneira ensinar e sim criar minha loja virtual para vender produtos de um velho sonho que tinha, sonho esse que era ter uma loja de produtos de informática e games (Seguindo os passos de meu pai, comerciante). Sempre via essa tal extensão .php nas urls dos sites e fóruns então resolvi que iria começar por aí, buscando saber o que era esse tal de PHP.

Alguém lembra das bancas de revistas, kkk. Fui a uma banca de revistas e comprei um mini-livro, da antiga Digerati, sobre PHP, entretanto, estava me precipitando muito pois não conhecia nada de lógica e o livro tinha aspectos avançados da linguagem, noob, achamos que tudo é a mesma coisa e penamos para encontrar o material correto rsrsrs!

Lembro de ter deixado o livro de lado e partido para entender como de fato começar e depois de muito penar e pesquisar fui pro zero absoluto, estudando lógica de programação e o funcionamento da web, então, o dia do livro chegou, quando finalmente consegui entender a proposta dele além de continuar a buscar mais conhecimentos para evoluir e montar minha loja, que cá entre nós tecnicamente nunca chegou sendo criada por mim...

Acabei que montando a loja mas por meio de serviços de loja pronta da época, entretanto, continuava a estudar programação e Web, me apaixonando cada vez mais! PHP foi minha linguagem de escolha e a

que me especializei e uso até hoje, inclusive, alcançando a tão sonhada certificação PHP.

No percurso e carreira, passando por empresas locais e trabalhos remotos encontrei o Laravel, na versão 3 ainda e de lá pra cá sempre tenho ele como uma ferramenta na minha bolsa de ferramentas quando preciso entregar aplicações rápidas e com muito valor.

Não acredito em bala de prata, entendo que existem casos e casos e a ferramenta ideal para o trabalho do momento!

Code Experts

A Code Experts é uma escola de cursos online com foco em prática onde abordamos os conceitos da programação web em projetos práticos e reais. A Code Experts, antes Code Experts Learning, começou como um braço de educação da Code Experts (Criação de Sistemas Web Sob Demanda).

Iniciamos na educação para suprir uma demanda faltante de cursos em nossa cidade, São Luis do Maranhão, e da vontade de transformar outras vidas por meio do ensino de programação, levando o conhecimento e simplificando o caminho para os devs iniciantes.

Dia após dia buscamos melhorar nossas metodologias de forma a gerar valor para nossos alunos resultando em melhoras, em nossos materiais e na didática abordada!

O trabalho com educação foi tão grande que resolvemos fazer o merge e unificar tudo em educação então a Code Experts Learning, de uma braço de educação, tornou-se o corpo todo! Tornou-se Code Experts - Cursos Online de Programação na Prática.

A loja virou a plataforma!

O conhecimento adquirido ao longo dos anos, que inicialmente seria para criar minha loja de produtos de informática e games tornou-se para criação de nossa plataforma de cursos online lançada em 2016, especificamente Novembro de 2016 e de lá pra cá venho concentrando meus esforços para a cada dia melhorar a experiência dos nossos alunos dentro dela.

O trabalho apenas começou!

Hoje sou desde o zelador ao editor de vídeos, passando por escritor e professor mas busco sempre dá o meu melhor para entregar excelentes materiais para nossos alunos!

Venha nos conhecer codeexperts.com.br.

Para quem é este livro?

Bom, serei breve aqui. Escrevi este livro para os programadores PHP e entusiastas que buscam melhorar sua produtividade na criação de Aplicações Web usando a linguagem, entregando softwares mais rápidos e com valor agregado. Nem sempre rápido quer dizer de má qualidade pois o Laravel te permite criar coisas com menos tempo e esforço mas com alto valor agregado e é por este caminho que irei te guiar nestas páginas.

É claro que se você apenas começou com PHP recomendo fortemente a continuar estudando as bases da linguagem e busque materiais sobre Orientação a Objetos pois estes conhecimentos irão te adicionar uma excelente base para entender todos os conceitos abordados, não somente no Laravel, mas em frameworks PHP modernos.

Se você está pronto para a jornada, então conte comigo e vamos lá!



Laravel 6

Laravel é um fullstack framework criado por Taylor Otwell. O Laravel foi criado em Junho de 2011 e de lá pra cá vêm ganhando muita notoriedade por meio de suas features e facilidades quando tratamos da criação de aplicações web e diversos serviços e softwares.

O Laravel foi fortemente inspirado pelo Rails, em sua forma de trabalho e gerações dentro da sua estrutura quando estamos no processo de desenvolvimento. O Laravel hoje chega em sua versão 6 e conta com um ecossistema em constante crescimento envolvendo diversos projetos e componenetes.

Conheceremos diversas funcionalidades do framework e veremos como ele traz essa simplicidade tão aclamada pela comunidade de desenvolvedores PHP.

Obtendo o Laravel

Podemos iniciar um projeto Laravel de duas maneira, a primeira é utilizando o Laravel Installer obtido através do composer. A outra é utilizando o proprio composer diretamente através do comando create-project.

Vamos conhecer ambas as formas, relambrando que para ambas precisamos ter o composer em nossa máquina. Para isso instale o mesmo através do site getcomposer.org.

Requisimentos

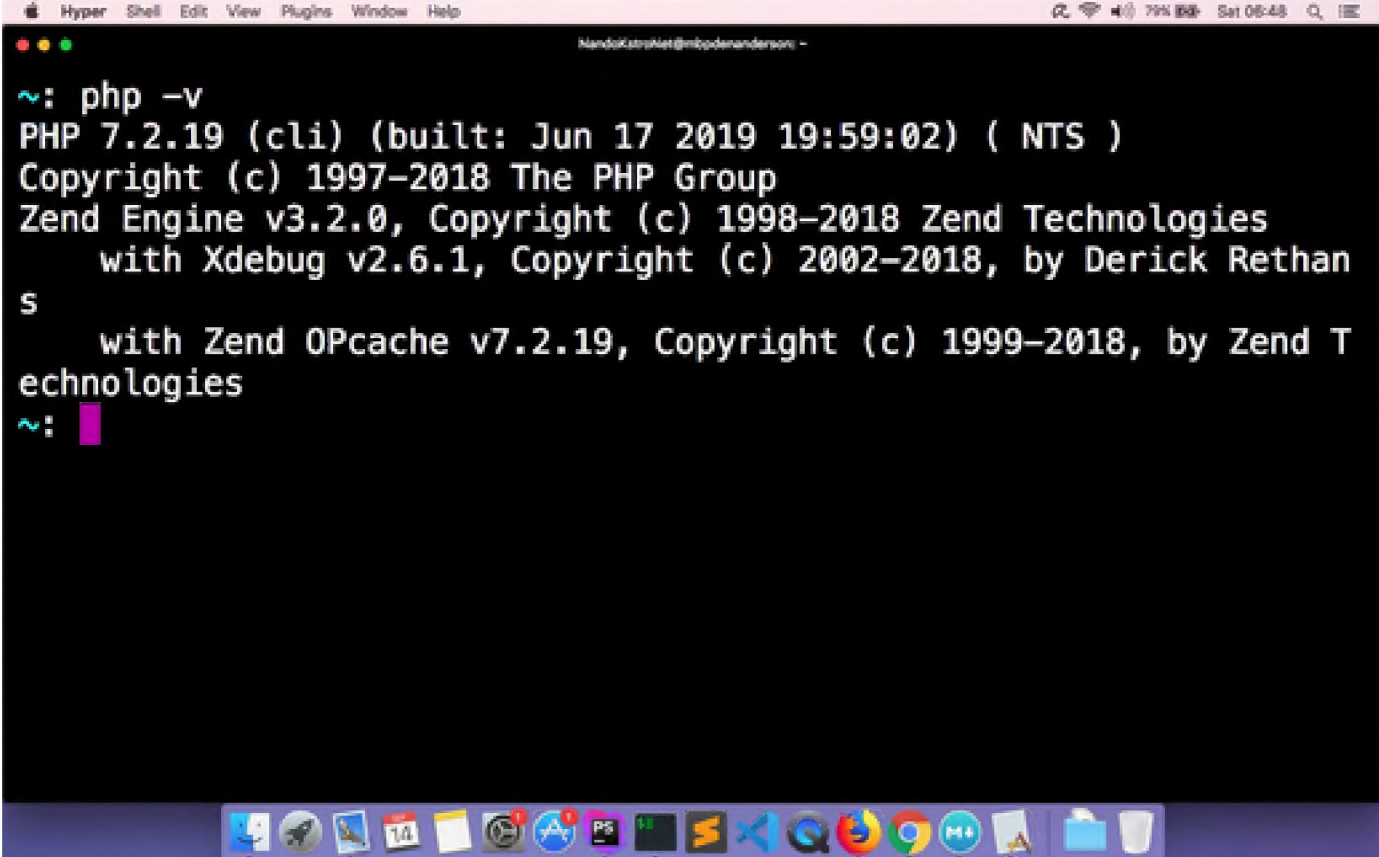
Para termos sucesso na utilização do Laravel verique se seu PHP respeita as seguintes configurações:

- PHP maior ou igual ao 7.2;
- Extensão BCMath

- Extensão Ctype
- Extensão JSON
- Extensão Mbstring
- Extensão OpenSSL
- Extensão PDO
- Extensão Tokenizer
- Extensão XML

Para verificar sua versão do PHP de forma rápida basta acessar seu terminal ou cmd(tenha certeza que o PHP está no PATH do Windows) e execute o comando abaixo:

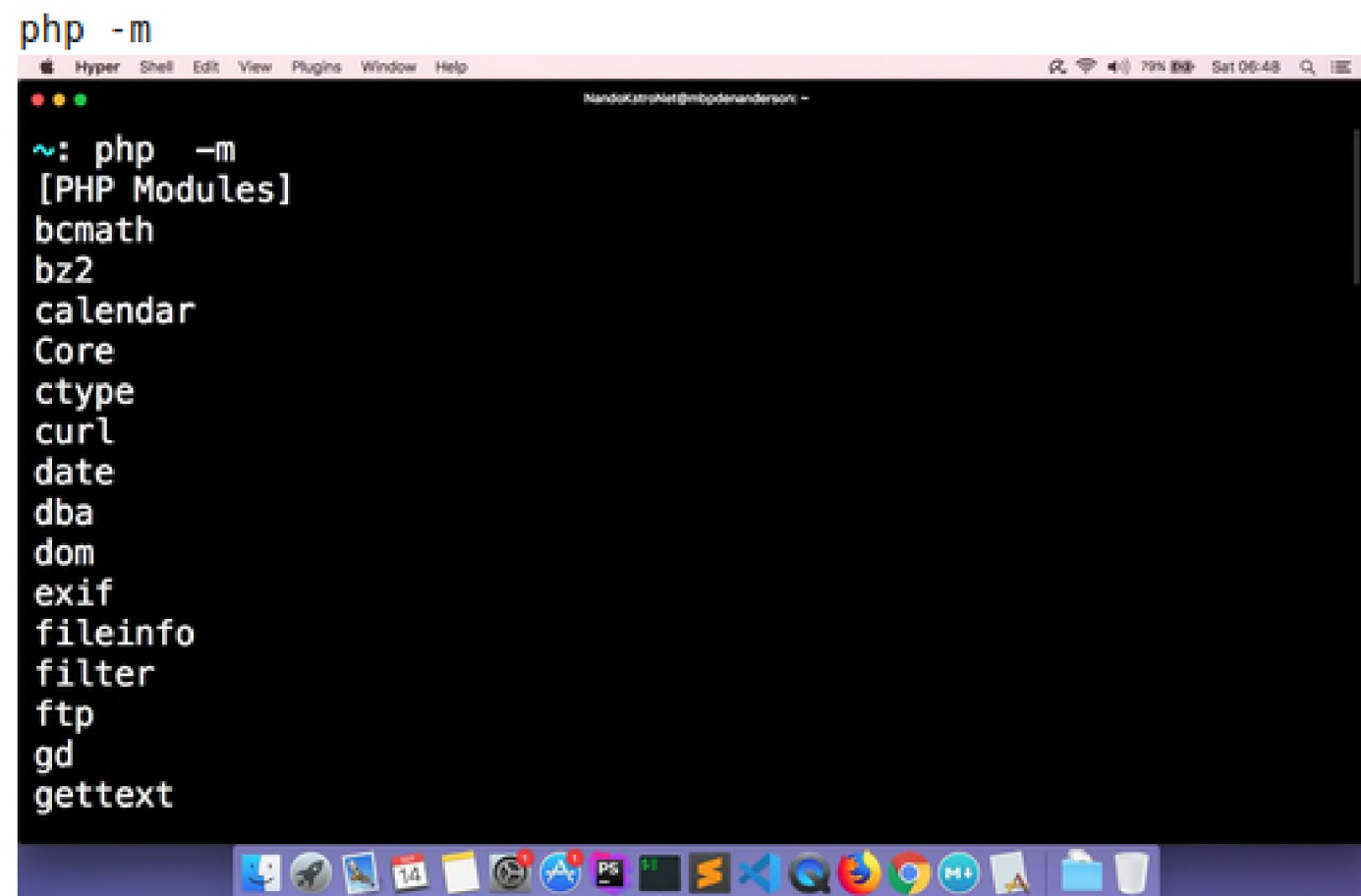
`php -v`



```

~: php -v
PHP 7.2.19 (cli) (built: Jun 17 2019 19:59:02) ( NTS )
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
    with Xdebug v2.6.1, Copyright (c) 2002-2018, by Derick Rethan
    with Zend OPcache v7.2.19, Copyright (c) 1999-2018, by Zend Technologies
~:
  
```

Para visualizar suas extensões execute em seu terminal o comand abaixo:



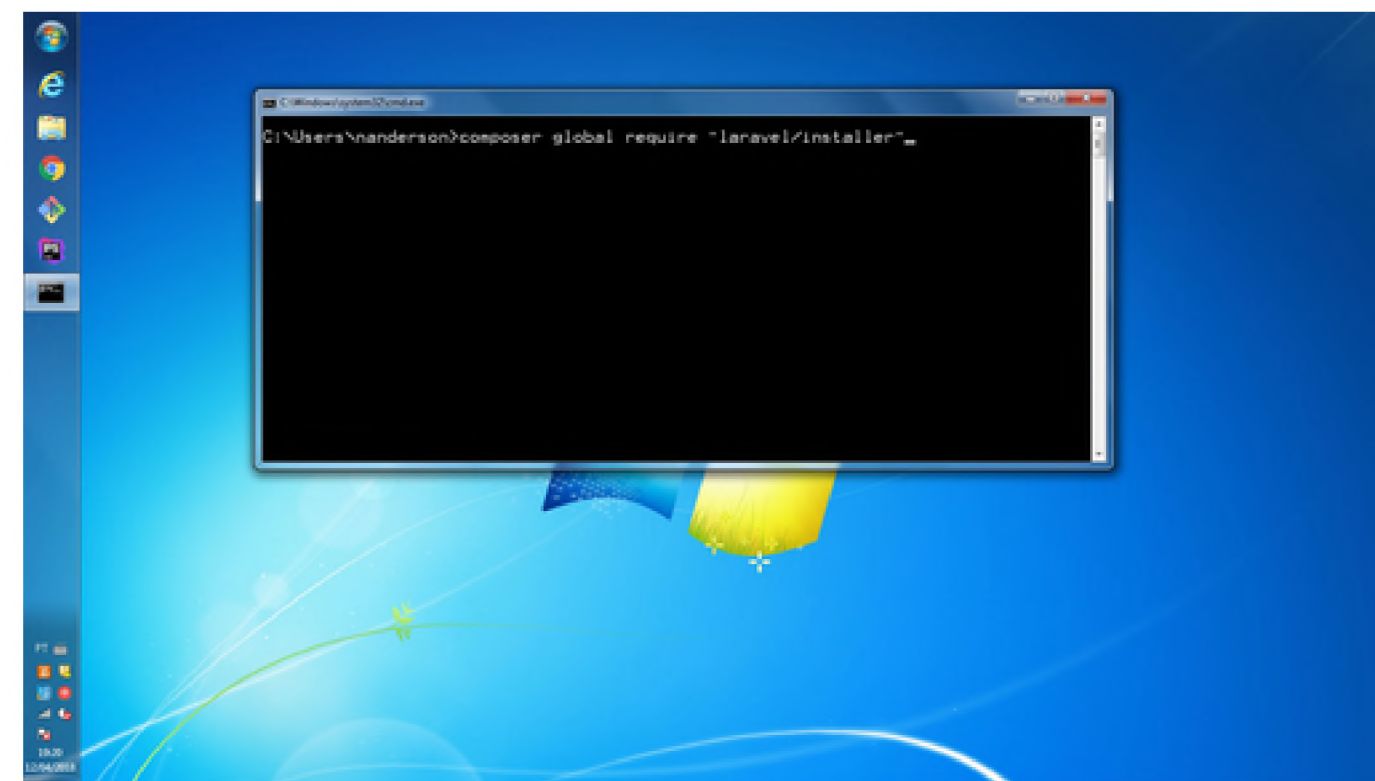
```
php -m
~: php -m
[PHP Modules]
bcmath
bz2
calendar
Core
ctype
curl
date
dba
dom
exif
fileinfo
filter
ftp
gd
gettext
```

Laravel Installer (Windows)

Com o composer em sua máquina Windows, vamos instalar o nosso utilitário, o Laravel Installer. Primeiramente abra seu prompt de comando, basta utilizar a combinação de teclas CTRL + R e na janelinha que aparecer digitar: cmd.

Após isso pressione ENTER para abrir o prompt.

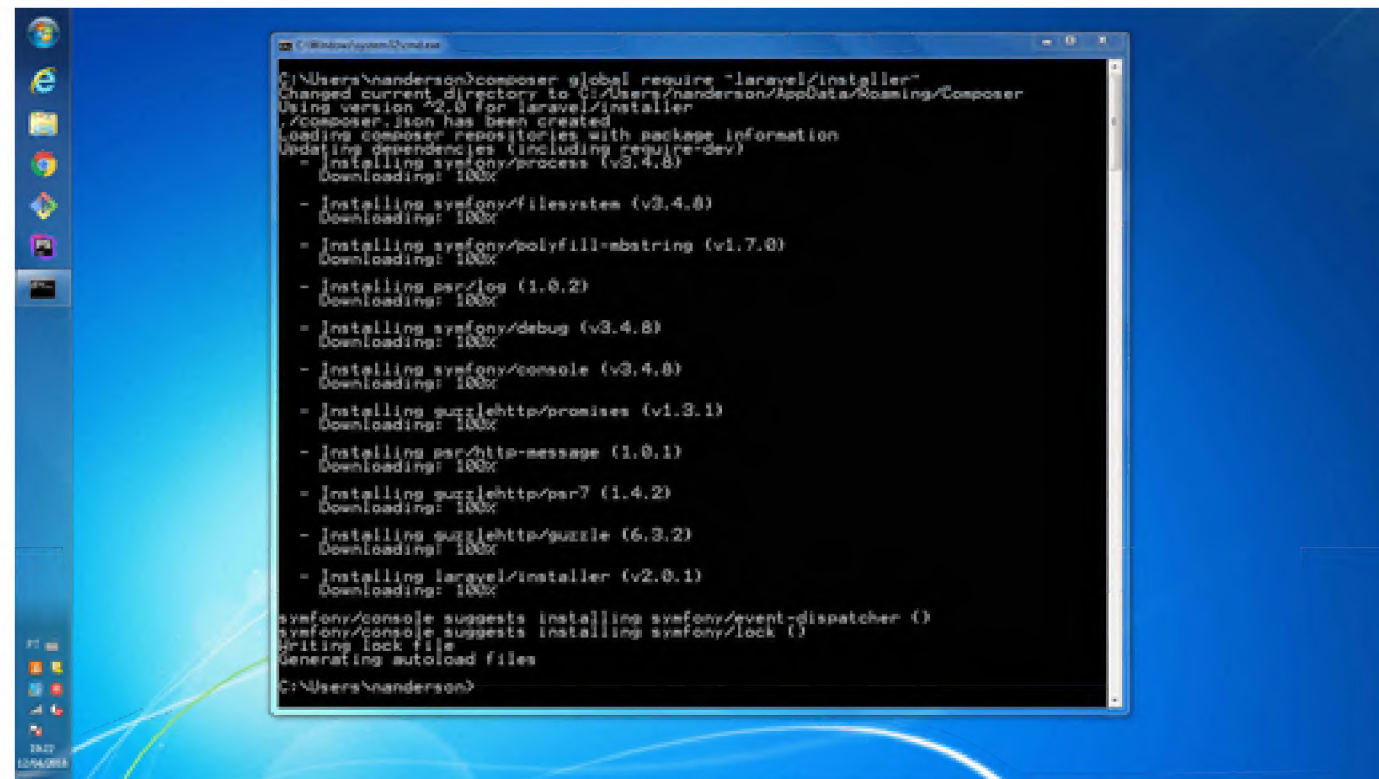
Com o prompt aberto, digite e execute o comando abaixo:
composer global require "laravel/installer"



Com isso o composer vai baixar o pacote laravel/installer e jogar na pasta abaixo, dentro de vendor & também dentro de bin, que está dentro de vendor, ambas estão no caminho abaixo:

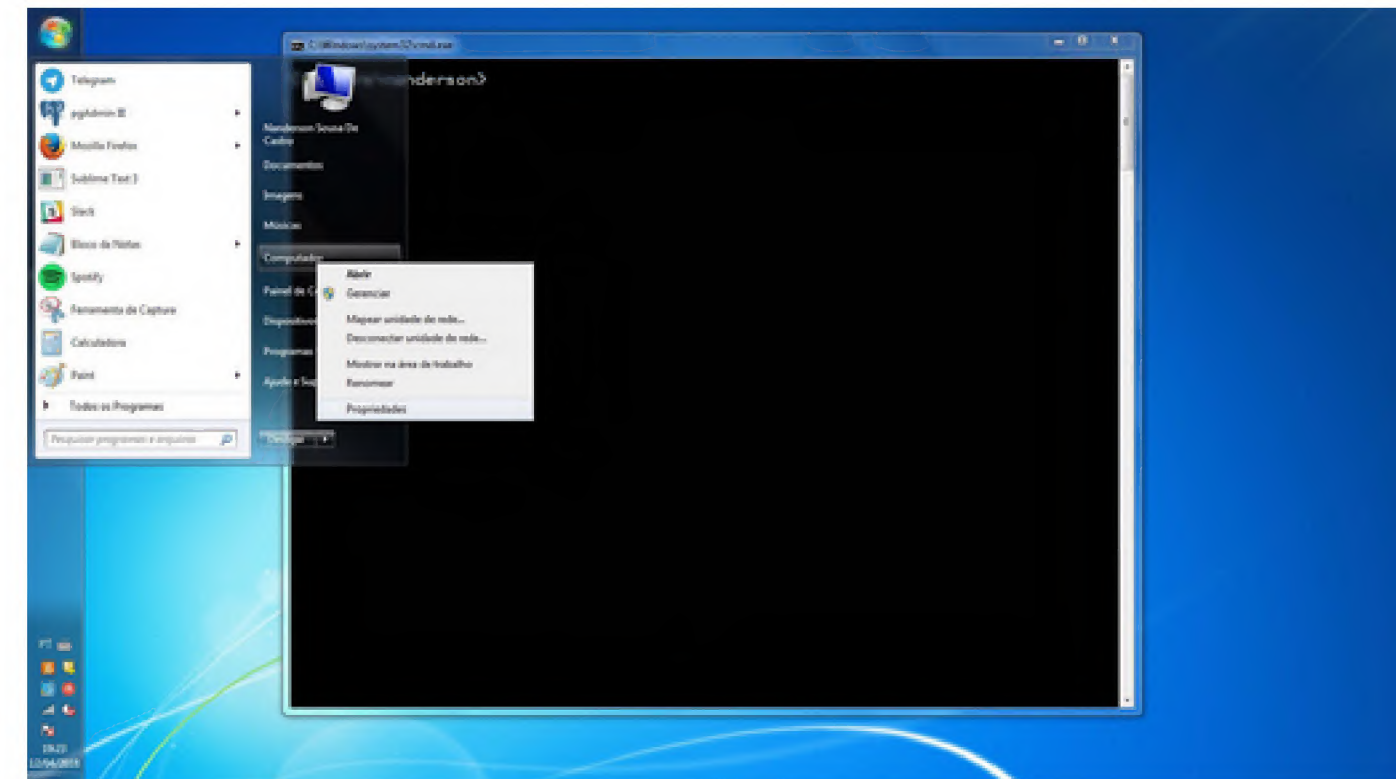
C:/Users/<seu_nome_de_usuario>/AppData/Roaming/Composer/

Veja o resultado da instalação abaixo:



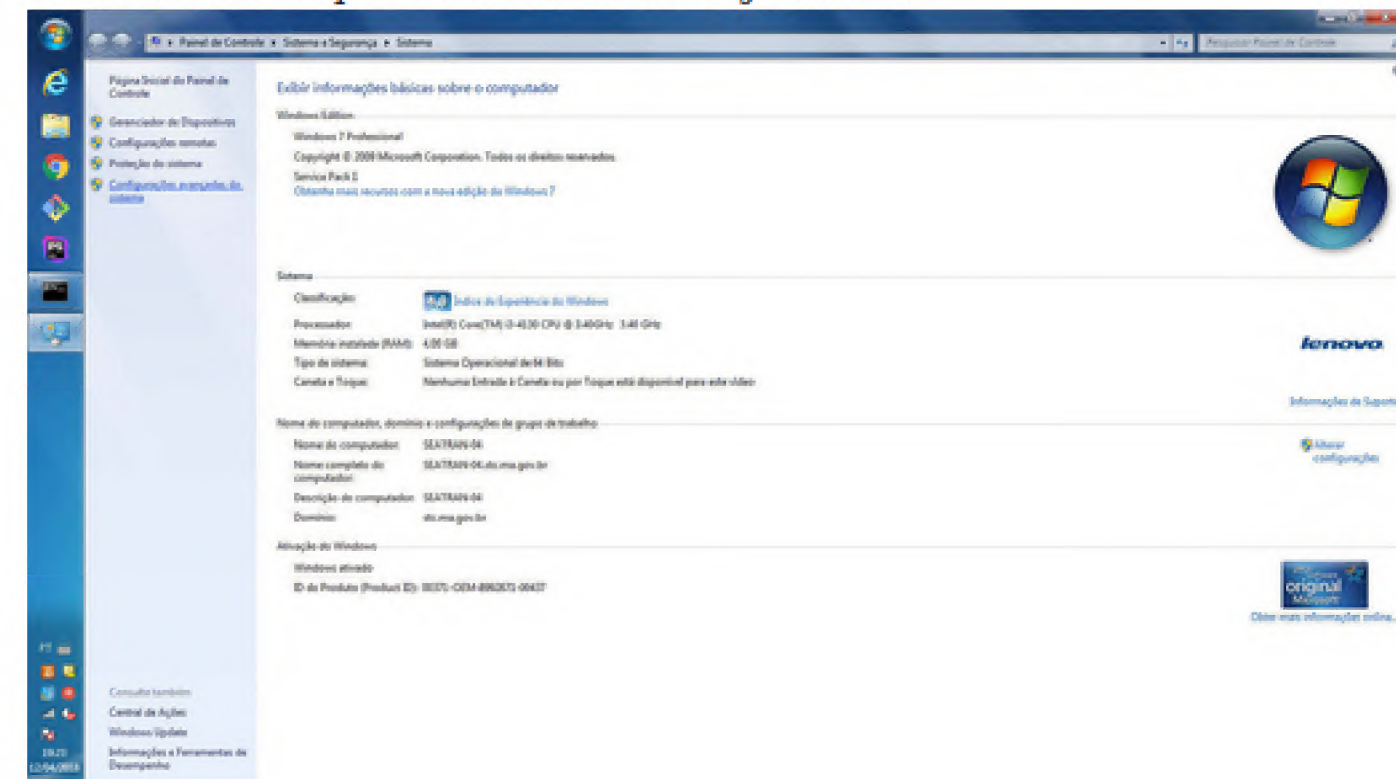
```
C:\Users\nanderson>composer global require "laravel/installer"
Changed current directory to C:\Users\nanderson\AppData\Roaming\Composer
Using version ^2.0 for laravel/installer
./composer.json has been created
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/process (v3.4.8)
  Downloading: 100%
- Installing symfony/filesystem (v3.4.8)
  Downloading: 100%
- Installing symfony/polyfill-mbstring (v1.7.0)
  Downloading: 100%
- Installing par/lev (1.0.2)
  Downloading: 100%
- Installing symfony/debug (v3.4.8)
  Downloading: 100%
- Installing symfony/console (v3.4.8)
  Downloading: 100%
- Installing guzzlehttp/promises (v1.3.1)
  Downloading: 100%
- Installing par/httplib (1.0.1)
  Downloading: 100%
- Installing guzzlehttp/par7 (1.4.2)
  Downloading: 100%
- Installing guzzlehttp/guzzle (6.3.2)
  Downloading: 100%
- Installing laravel/installer (v2.0.1)
  Downloading: 100%

symfony/console suggests installing symfony/event-dispatcher ()
symfony/console suggests installing symfony/lock ()
Writing lock file
Generating autoload files
C:\Users\nanderson>
```

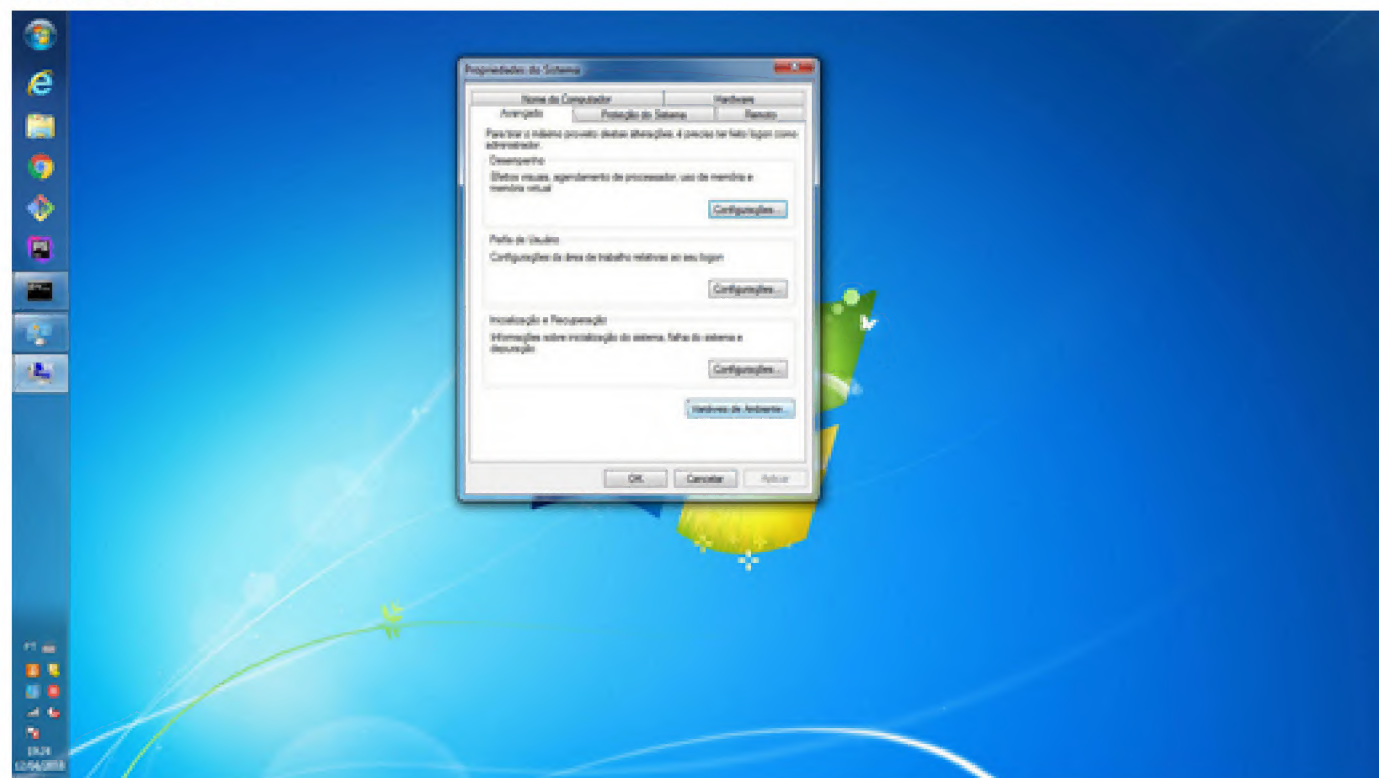


Feito isso, precisamos adicionar o caminho dos binários globais do composer no PATH do Windows. Primeiramente acesse o menu propriedades do Meu Computador, como mostrado abaixo:

Após isso, acesse o menu Configurações Avançadas do Sistema, no menu lateral esquerdo da sua tela. Veja abaixo:

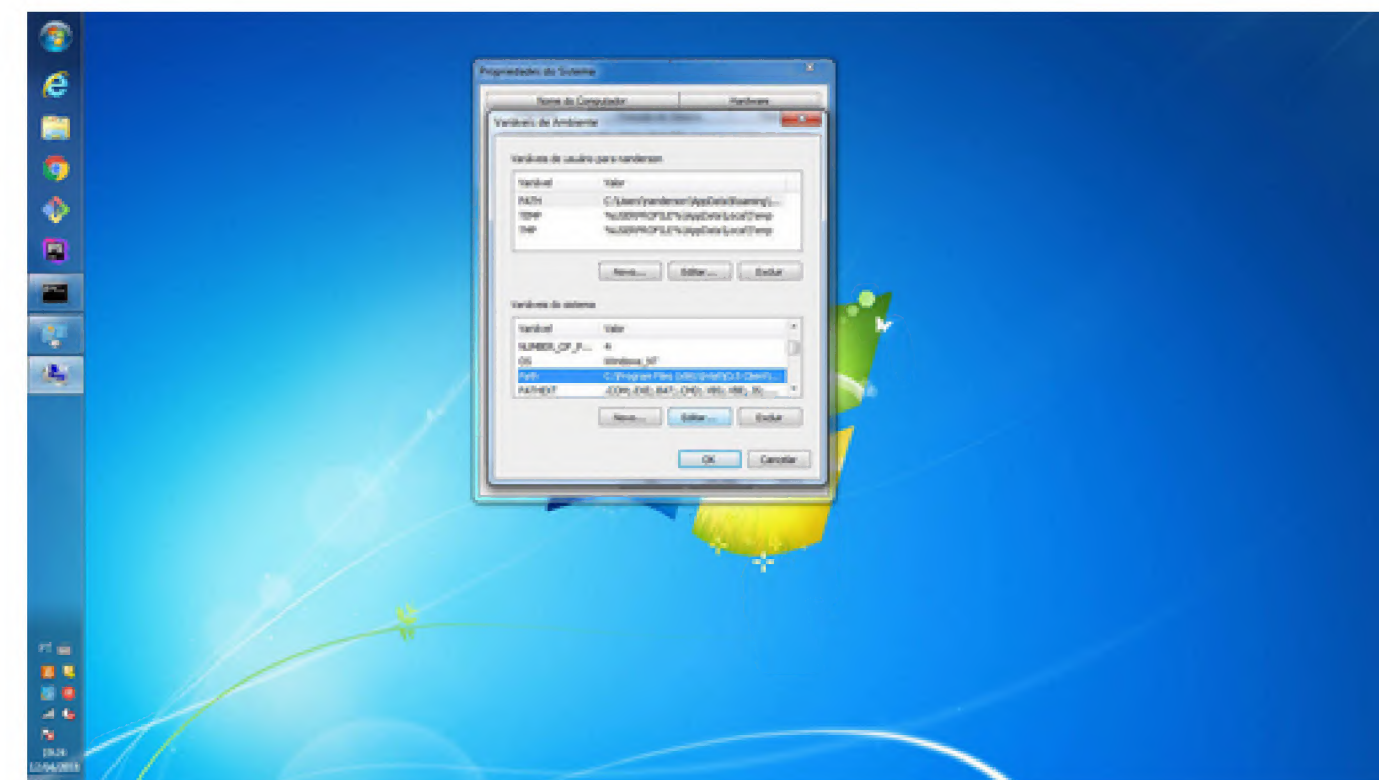


Na tela que aparecer, na parte inferior, acesse o menu Variáveis de Ambiente*.



Se você está em um ambiente com Windows 10, pode ser que a tela a seguir mude um pouco. Na verdade fica mais simples de adicionar um caminho ao PATH do Windows 8 pra frente.

Na janela de Variáveis de Ambiente, na parte inferior, procure por Path e clique em editar. Como mostro abaixo:



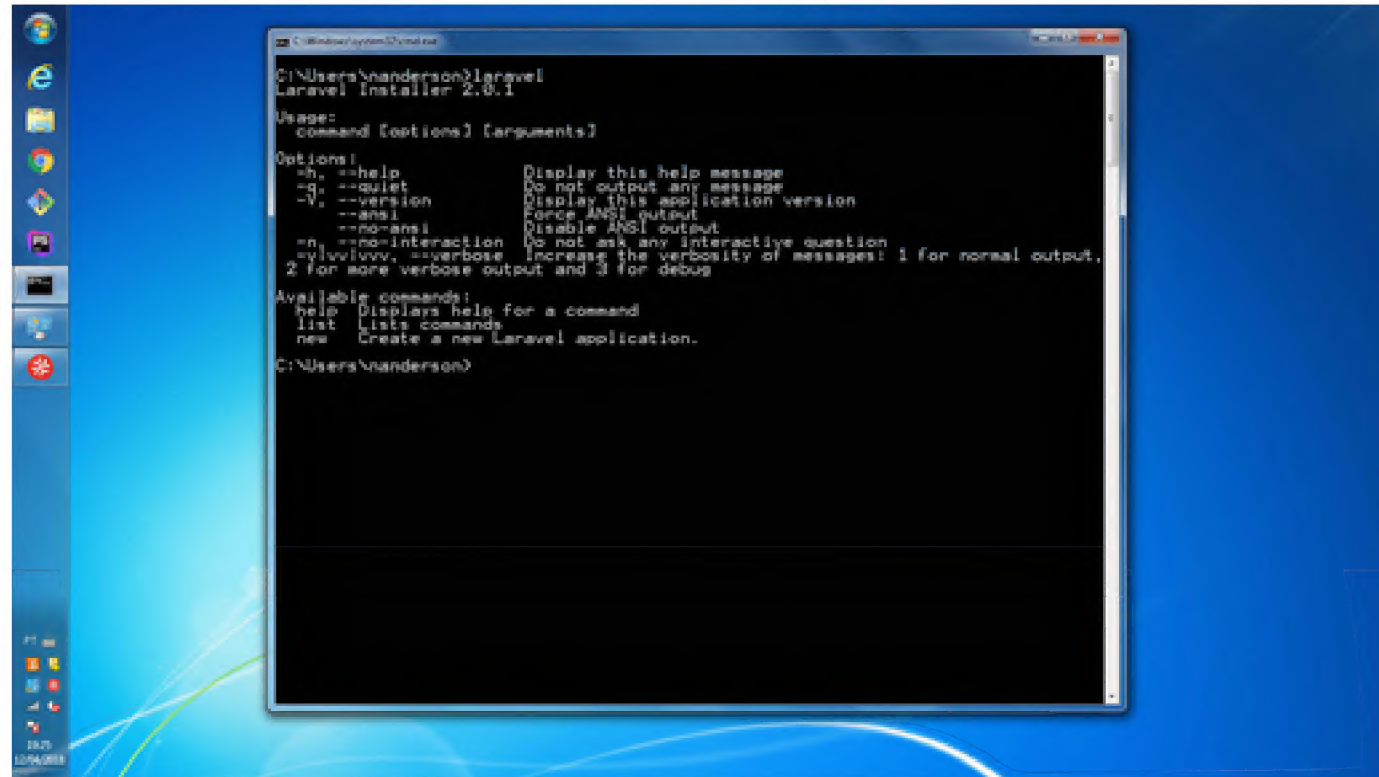
Na janelinha que aparecer, modifique o valor da variável. Neste caso apenas adicione ao final da linha o caminho abaixo substituindo o <seu_nome_de_usuario> por seu nome de usuário no sistema:
%USERPROFILE%/AppData/Roaming/Composer/vendor/bin;

Obs.: Verifique se ao fim da linha existe um ;(ponto-e-virgula), caso não exista adicione um e depois coloque o caminho indicado acima. Tome cuidado para não remover o conteúdo da variável Path, apenas adicione o caminho a mais.

PS.: O ponto-e-virgula que você vê ao fim do caminho acima é o separador de paths no Windows, para a variável PATH do sistema. Por isso é importante verificar se ao fim do valor que já existe, há um ponto-e-virgula.

Após isso, dê OK até sair de todas as janelas. Se seu prompt continuou aberto durante o processo, feche e abra novamente.

Para verificar se o laravel installer é reconhecido, execute em seu cmd o comando: laravel e obtenha o resultado abaixo:



```
C:\Users\wander>laravel
Laravel Installer 2.0.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
  --ansi               Force ANSI output
  --no-ansi            Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  -vvvvv, --verbose     Increase the verbosity of messages: 1 for normal output,
                        2 for more verbose output and 3 for debug

Available commands:
  help  Displays help for a command
  list  Lists commands
  new   Create a new Laravel application.

C:\Users\wander>
```

Agora estamos aptos a iniciarmos um projeto Laravel por meio do Laravel Installer em nosso CMD.

Laravel Installer (MacOS ou Linux / Unix Like)

Tendo a certeza que o Composer está em sua máquina, execute o comando abaixo em seu terminal:

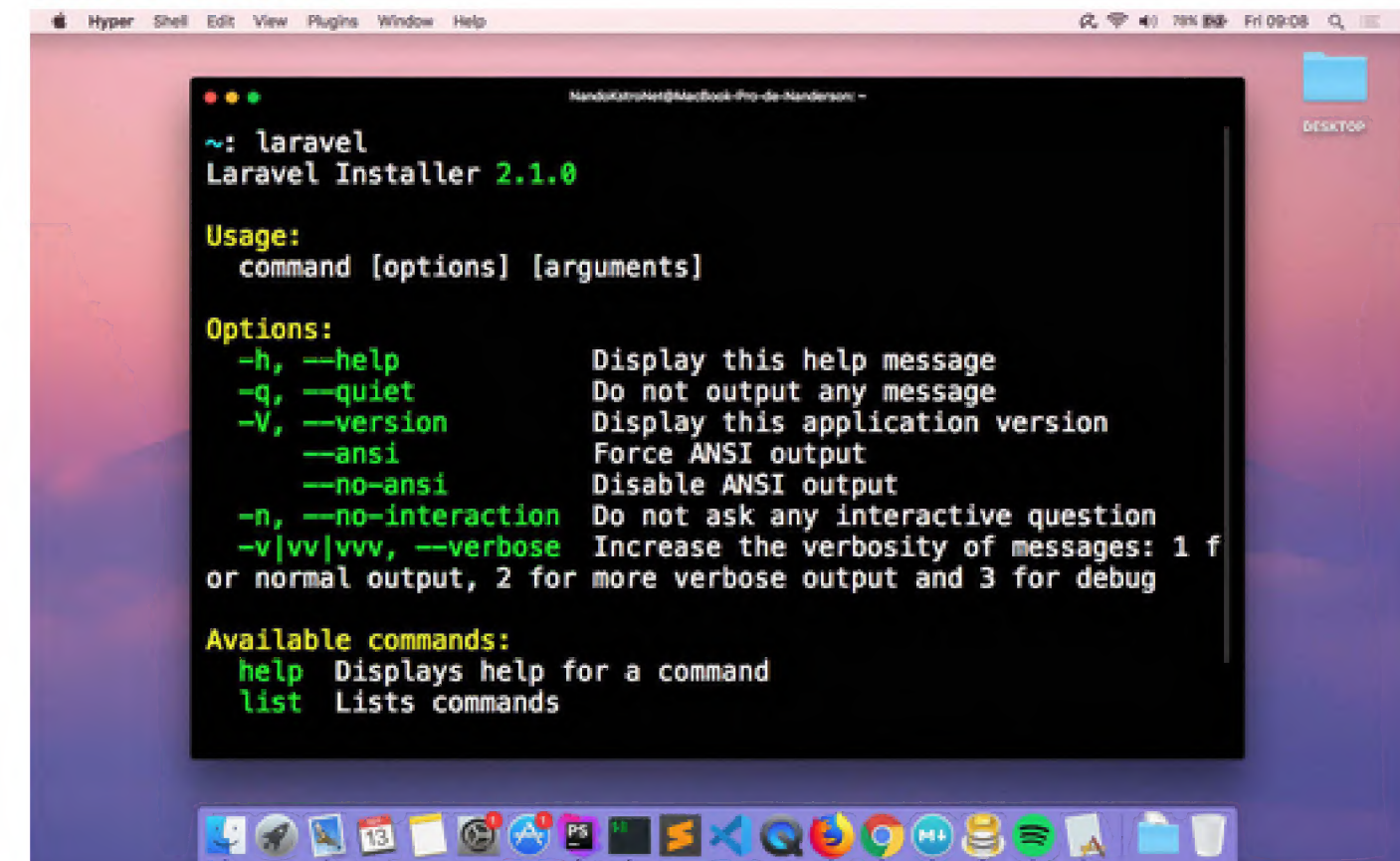
`composer global require "laravel/installer"`

Após a instalação do pacote precisamos linkar ele no PATH do nosso sistema. Se você usa o bash em seu terminal o arquivo para que você possa configurar o PATH será o `.bash_profile`, caso utilize o zsh o arquivo a ser alterado será o `.zshrc`. Ambos os arquivos encontram-se na pasta do seu usuário. Pelo terminal você pode digitar `cd ~` e dá um enter que cairá na pasta do seu usuário pelo seu terminal.

Ao abrir o arquivo correspondente, adicione a seguinte linha ao final do arquivo:

`PATH="$HOME/.composer/vendor/bin:$PATH"`

Após isso reinicie seu terminal e execute o comando laravel, e obtenha o resultado abaixo:



```
~: laravel
Laravel Installer 2.1.0

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
  --ansi               Force ANSI output
  --no-ansi            Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 f
                        or normal output, 2 for more verbose output and 3 for debug

Available commands:
  help  Displays help for a command
  list  Lists commands
```

Iniciando Primeiro Projeto

Iniciando projeto com Composer Create Project

Podemos iniciar um projeto laravel por meio do comando `create-project` do composer sem necessariamente usar o laravel installer.

Para isso, para iniciar um projeto laravel por meio deste comando basta executa-lo como mostro abaixo, na pasta da sua escolha:


```
composer create-project --prefer-dist laravel/laravel blog
```

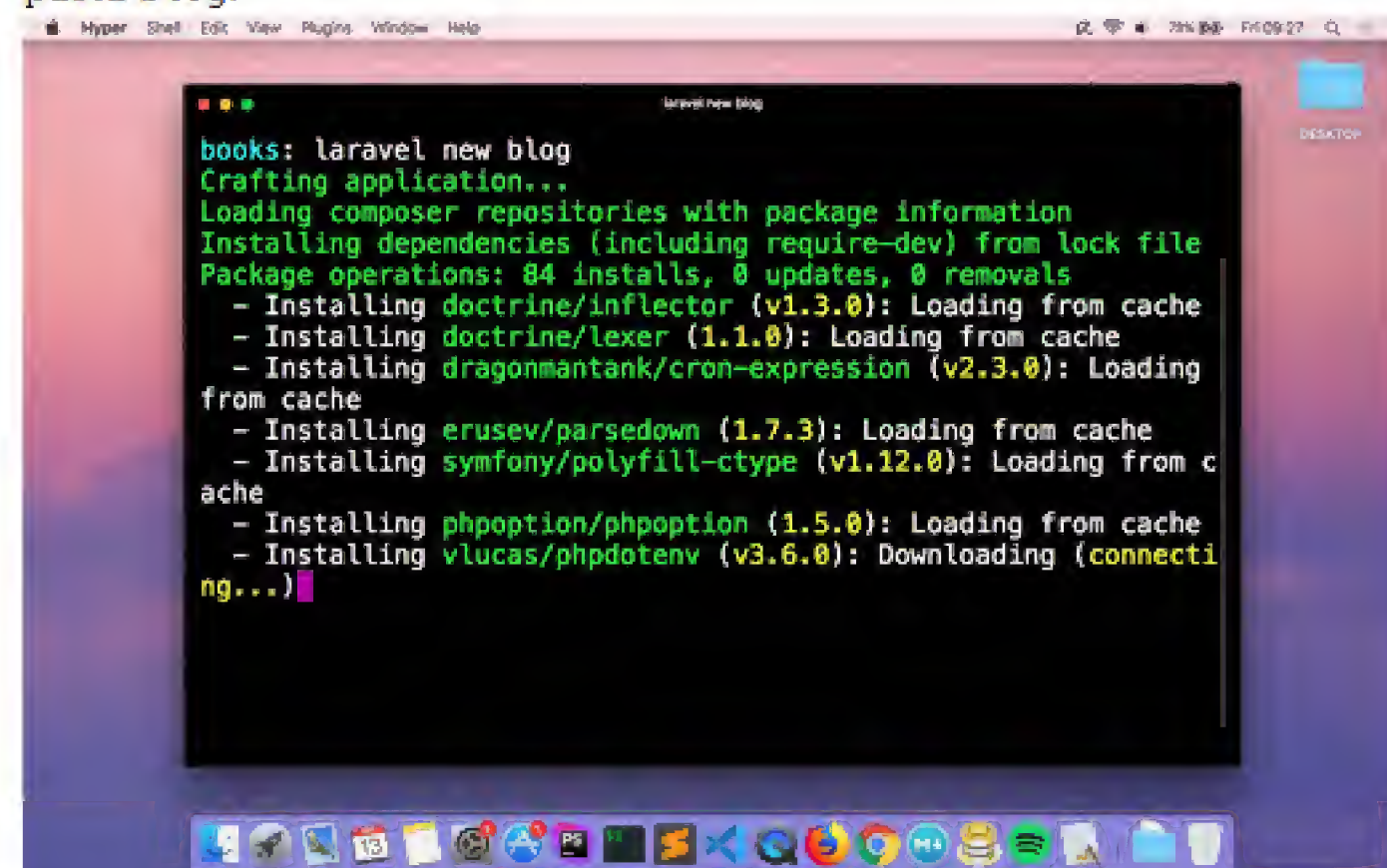
Com isso o instalador baixará todo o skeleton do Laravel e instalará as dependências do nosso projeto automaticamente e jogará dentro da pasta blog.

Iniciando projeto com Laravel Installer

Podemos iniciar nosso projeto por meio do laravel installer também então para isso basta acessarmos a pasta de escolha pelo terminal e executarmos o comando abaixo:

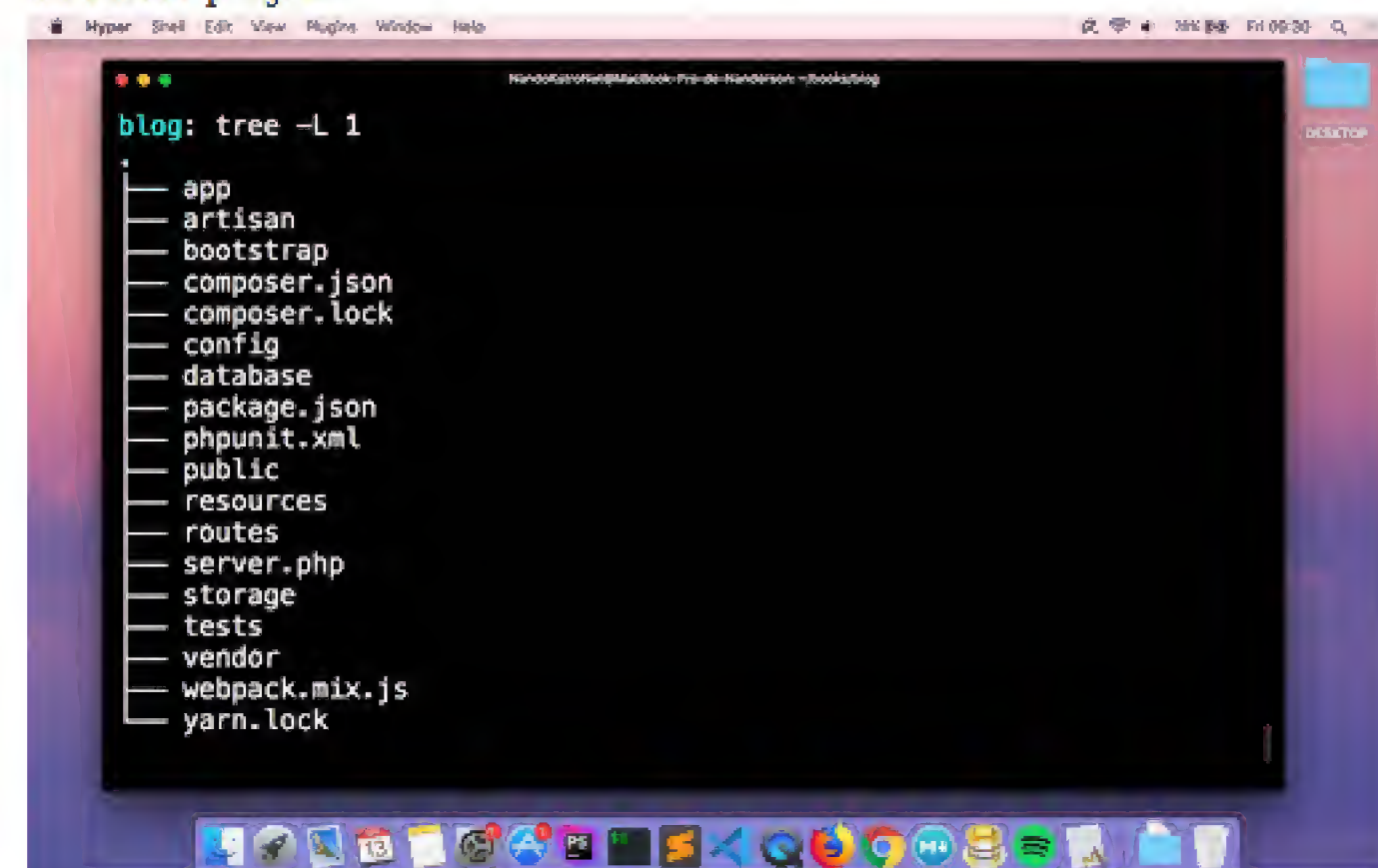
```
laravel new blog
```

Com isso o instalador baixará todo o skeleton do Laravel e instalará as dependências do nosso projeto automaticamente e jogará dentro da pasta blog.



Conhecendo a estrutura do Laravel

Vamos conhecer as pastas e os arquivos que fazem parte da estrutura do nosso projeto:



O Laravel possui algumas pastas base como a pasta app, bootstrap, database, config, public, resources, storage, routes, tests e a vendor. Vamos ver o que cada pasta destas representa ou armazena em sua estrutura.

app

A pasta app conterá todo o conteúdo do nosso projeto como os models, controllers, serviços, providers, middlewares e outros. Nela que concentraremos diretamente nossos esforços durante a criação do nosso projeto.

config

Os arquivos de configurações do nosso projeto laravel encontram-se nesta pasta. Configurações de conexões com bando, onde estão os drivers para armazenamento de arquivos, configurações de autenticação, mailers, serviços, sessions e outros.

resources

Nesta pasta temos algumas subpastas que são: views, js, lang & sass. A priori esta pasta salva os assets referentes as nossas views e também nossas views ou templates.

storage

Nesta pasta salvamos arquivos como sessions, caches, logs e também é utilizada para armazenar arquivos mediante upload em nosso projeto.

routes

Nesta pasta vamos encontrar os arquivos para o mapeamento das rotas de nosso projeto. Rotas estas que permitirão o nosso usuário acessar determinada url e ter o conteúdo processado e esperado. Mais a frente vamo conhecer melhor essas rotas mas as mesmas são divididas em rotas de api, as rotas padrões no arquivo web, temos ainda rotas para channels e console.

tests

Nesta pasta teremos as classes para teste de nossa aplicação. Testes Unitários, Funcionais e outros.

database

Aqui teremos os arquivos de migração de nossas tabelas, vamos conhecer eles mais a frente também, teremos os arquivos para os seeds e também as factories estes para criação de dados para popularmos nossas tabelas enquanto estamos desenvolvendo.

bootstrap

Na pasta bootstrap teremos os arquivos responsáveis por inicializar os

participantes do framework Laravel, ecaminhando as coisas a serem executadas.

public

Esta é nossa pasta principal, a que fica exposta para a web e que contém nosso front controller. Por meio desta pasta é que recebemos nossas requisições, especificamente no index.php, e a partir daí que o laravel direciona as requisições e começa a executar as coisas.

vendor

A vendo como conhecemos, é onde ficam os pacotes de terceiros dentro de nossa aplicação mapeados pelo composer.

Outros arquivos da raiz do projeto

Temos ainda alguns arquivos na raiz do nosso projeto, como o `composer.json` e o `composer.lock` onde estão definidas as nossas dependências e as versões baixadas respectivamente.

Temos também o `package.json` que contém algumas definições de dependências do frontend. Temos também os arquivos de configuração para o webpack, pacote responsável por criar os builds do frontend.

Temos ainda também o `server.php` que nos permite emular o `mod_rewrite` do apache.

Temos também o `phpunit.xml` que contém as configurações para nossa execução dos testes unitários, funcionais e etc em nossa aplicação.

Deixe por último o arquivo `.env` que contém as variáveis de ambiente para cada configuração de nossa aplicação como os parâmetros para conexão com o banco e também o `application key hash` único para nossa aplicação e outras configurações a mais além destas.

Este arquivo e essas configurações são providas pelo pacote DotEnv do Vance Lucas.

Laravel: Artisan CLI

O Laravel possui uma interface de comandos ou command line interface (CLI) chamada de artisan. Por meio dela podemos melhorar bastante nossa produtividade enquanto desenvolvemos como por exemplo: Gerar models, controllers, gerar a interface de autenticação e muitas outras opções que conheceremos ao longo do livro. Para conhecer todos os comando disponíveis no Artisan, basta executar na raiz do seu projeto o seguinte comando:

```
php artisan
```

Veja o resultado, a lista de comandos e opções disponíveis no cli:

```
HandelKiroHag@MacBook-Pro-de-Handerson: ~/books/blog
blog: php artisan
Laravel Framework 6.9.3

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet            Do not output any message
  -V, --version          Display this application version
  --ansi                Force ANSI output
  --no-ansi             Disable ANSI output
  -n, --no-interaction   Do not ask any interactive question
  --env[=ENV]           The environment the command should run under
  -vv|vvv, --verbose    Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  clear-compiled  Remove the compiled class file
  down           Put the application into maintenance mode
  env            Display the current framework environment
  help          Display help for a command
  inspire       Display an inspiring quote
  list          List commands
  migrate       Run the database migrations
  optimize      Compile the framework bootstrap files
  preset        Swap the front-end scaffolding for the application
  serve         Serve the application on the PHP development server
  tinker        Interact with your application
  up           Bring the application out of maintenance mode
  auth
  auth:clear-resets  Flush expired password reset tokens
  cache
  cache:clear       Flush the application cache
  cache:forget      Remove an item from the cache
  cache:table       Create a migration for the cache database table
  config
  config:cache      Create a cache file for faster configuration loading
  config:clear      Remove the configuration cache file
  db
  db:seed           Seed the database with records
  down             Drop all tables, views, and types
  event
  event:cache       Discover and cache the application's events and listeners
  event:clear       Clear all cached events and listeners
  event:generate     Generate the missing events and listeners based on registration
```

Executando a Aplicação

Para concluirmos nosso primeiro capítulo, vamos iniciar nossa aplicação

e testá-la em nosso browser. Para isso acesse o seu projeto via terminal ou cmd no Windows e na raiz execute o comando abaixo:

```
php artisan serve
```

```
blog: php artisan serve
Laravel development server started: <http://127.0.0.1:8000>
```

O comando acima levantará sua aplicação em seu ip local 127.0.0.1 e disponibilizará a porta 8000 para que você possa acessar sua aplicação no browser. Agora vamos acessar em nosso browser o seguinte link: <http://127.0.0.1:8000>:



Se tudo estiver corretamente configurado teremos o resultado acima, a tela inicial do nosso projeto Laravel, e sua página inicial default.

Conclusões

Bom neste módulos concluímos aqui as configurações e realizamos o início do nosso primeiro projeto utilizando o framework. Agora vamos continuar para o próximo capítulo e começar a entender a estrutura geral do framework por meio da criação do nosso Hello World utilizando o framework.

Hello World com Laravel

Este capítulo visa mostrar o Laravel de uma forma geral, o que definirá nosso fluxo de trabalho durante todo o desenvolvimento. Gosto muito de tomar uma abordagem prática, por isso, vamos trabalhar todos os conceitos envolvendo o framework em cima de um projeto prático e direto ao ponto.

O projeto escolhido e que nos traz todo o aparato para entendermos cada parte de um framework fullstack será a criação de um blog com gerenciamento de posts e autores, além do sistema de comentários. Então vamos lá colocar a mão na massa e dar início de fato ao conhecimento por meio do nosso primeiro Hello World com o framework.

MVC

Antes de continuarmos precisamos conhecer um pouco do modelo base utilizado no mapeamento das classes dentro do nosso projeto Laravel, esse modelo ou padrão é o famoso MVC, ou, Model-View-Controller.

A maioria dos frameworks atuais utilizam esta estrutura para organização de seus componentes, além de termos um Front Controller que recebe as requisições enviadas para nossa aplicação e entrega/delega para quem vai resolver aquela requisição. Geralmente este Front Controller se encontra na pasta public(Directory Root da aplicação) no index.php dos frameworks, como o é no Laravel.

O modelo MVC possui três camadas base, as que comentei acima, o Model, o Controller e a View. Vamos entender o que é cada parte:

Model

A camada do Model ou Modelo é a camada que conterá nossas regras

de negócio, geralmente possuem as entidades que representam nossas tabelas no banco de dados, podem conter classes que contêm regras de negócio específicas e até podem conter classes que realizam algum determinado serviço.

Dentro do Laravel nossos models serão as classes que representam alguma tabela no banco de dados com poderes de manipulação referentes a esse tratamento com o banco. Você pode encontrar, por default, as classes model dentro da pasta app em sua raiz.

Controller

A camada do Controller ou Controlador é a camada mais fina digamos assim, ele recebe a requisição e demanda para o models em questão, caso necessário, e dada a resposta do model entrega o resultado para a view ou carrega diretamente um view caso não necessitemos de operações na camada do Model.

A ideia pro controller é que ele seja o mais simples possível, portanto, evite adicionar regras e complexidade em seus Controllers. Dentro do Laravel estes controllers encontram-se na pasta app/Http/Controllers.

View

A camada de View ou visualização é a camada de interação do usuário. Onde nossos templates vão existir, com as telas do nosso sistema e as páginas de nossos sites. Nesta camada, também, não é recomendado colocar regra de negócios, consultas ao banco ou coisas deste tipo. Esta camada é exclusivamente para exibição de resultados e input de dados apenas, via formulários, além de interações Javascripts e outros processos já esperados para melhor aproveitamento do usuário.

No Laravel nossas views encontram-se na pasta resources/views/. Em nossas views utilizaremos o template engine, que falarei sobre ele mais a frente, chamado de **Blade**.

Roteiro para nosso primeiro Hello World

Para criarmos nosso hello world, precisaremos seguir os passos abaixo para este momento:

- Criar um controller para execução ao chamarmos nossa rota em questão;
- Criar uma view para envio da nossa string: Hello World a ser exibida como resultado do acesso;
- Criar nossa rota para chamada e acesso em nosso browser.

A priori passos bem simples e que vão nos dar um panorama inicial do framework para a partir daí prosseguirmos com os conceitos individualmente.

Iniciando Hello World

Acesse o projeto iniciado no capítulo passado pelo seu terminal ou cmd no Windows. Na certeza de está na raiz do seu projeto execute o seguinte comando abaixo:

```
php artisan make:controller HelloWorldController
```

Ao executar o comando acima teremos o seguinte resultado, pro sucesso da criação do nosso primeiro controller, exibido em nosso terminal ou cmd: Controller created successfully.

Após isso teremos o nosso controller adicionado dentro da pasta do nosso projeto, especificamente: app/Http/Controllers e o arquivo HelloWorldController.php. Não esqueça de acessar o projeto em seu editor ou IDE de código de sua preferência.

O código do nosso controller encontra-se abaixo:

```
1 <?php
2
```



```

3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 class HelloWorldController extends Controller
8 {
9     //
10 }

```

O resultado acima é a classe do nosso controller que estende do Controller base e já traz um import para nós, o nosso Request que também conheceremos ele no decorrer do nosso livro.

Vamos criar nosso primeiro método para execução posterior ao acessarmos nossa rota, ainda não criada. Abaixo segue o conteúdo do nosso primeiro método:

```

1 public function index()
2 {
3     $helloWorld = 'Hello World';
4
5     return view('hello_world.index',
compact('helloWorld'));
6 }

```

Do método acima temos um ponto bem interessante, o retorno da função helper chamada view que recebe como primeiro parâmetro a view desejada e o segundo parâmetro um array associativo com os valores a serem enviados para esta view.

Antes de entendermos como criaremos a view com base no primeiro parâmetro, preciso te falar que a função compact é uma função do PHP que pega as variáveis informadas e joga dentro de um array associativo sendo a chave o nome da variável e seu valor o valor contido na variável, foi o que fizemos para enviar \$helloWorld para nossa view.

Agora como podemos criar nossa view de forma que o primeiro parâmetro da função helper view seja satisfeito? Vamos entender!

Temos o seguinte valor: `hello_world.index`. A última parte da string apresentada pós `.` será o nome da nossa view que deve respeitar o seguinte nome de arquivo, se depois do ponto tenho index precisarei criar uma view chamada `index.blade.php`, só que antes do ponto temos o valor `hello_world` que neste caso será a pasta onde nosso `index.blade.php` estará, então, no fim das contas precisaremos criar lá dentro da pasta `resources/views` uma pasta chamada `hello_world` e dentro desta pasta nosso arquivo `index.blade.php` chegando ao caminho completo e o arquivo: `resources/views/hello_world/index.blade.php`.

PS.: Se você quiser chamar uma view diretamente que esteja dentro da pasta de views, basta informar apenas o nome da views em questão. Se você tiver mais níveis em questão de pastas até chegar na view, é necessário informá-los até o arquivo da view em questão. Tanto o caminho base até a pasta views quanto a extensão `.blade.php` o próprio Laravel adiciona automaticamente para nós.

Com isso crie o arquivo `index.blade.php` e sua pasta `hello_world` dentro da pasta views. Com o seguinte conteúdo abaixo:

```
1 <h1>{{ $helloWorld }}</h1>
```

Acima temos, dentro do elemento `h1`, o nosso primeiro contato com o template engine Blade. Usamos acima a notação de print, abre `{{` fecha `}}`, dentro da nossa view e pegamos a nossa variável `$helloWorld` vinda lá do nosso controller e exibimos seu valor dentro do elemento html.

Agora, que já seguimos os dois passos do nosso roteiro precisamos permitir o acesso e execução deste método, método `index` do nosso controller `HelloWorldController`, por parte dos nossos usuários e a

liberação de uma url para acesso.

Como faremos isso? Simples, no momento, vamos criar uma rota que apontará para o método do nosso controller! Vamos lá que vou te mostrar!

Abra seu arquivo `web.php` que se encontra na pasta `routes` na raiz do projeto.

Teremos o seguinte conteúdo:

```
1 <?php
2
3 /*
4
5 |-----
6
7 | Web Routes
8 |-----
9
10 | Here is where you can register web routes for your
    application. These
11 | routes are loaded by the RouteServiceProvider within a
    group which
12 | contains the "web" middleware group. Now create something
    great!
13 |
14 */
15 Route::get('/', function () {
16     return view('welcome');
17 });
```

Este arquivo `web.php` conterá todas as rotas da nossa aplicação que trabalham com a exibição de `htmls` com resultados em nosso projeto. Tudo que for UI ou User Interface(Interface do Usuário) terá suas rotas definidas neste arquivo.

De cara já vemos a primeira definição de rota, a rota principal `/` que exibe a view `welcome.blade.php` que está lá dentro da pasta de `views`. Esta rota é a rota executado ao acessarmos a página principal de uma aplicação Laravel recém instalada. Agora vamos definir nossa rota de `hello world`.

Adicione o código abaixo, após a definição da rota principal que já existe:

```
1 Route::get('hello-world', 'HelloWorldController@index');
```

Acima temos nossa primeira rota definida, a rota escolhida foi `hello-world` que executará o método `index` do controller `HelloWorldController`, blz, como isto está definido? Vamos lá!

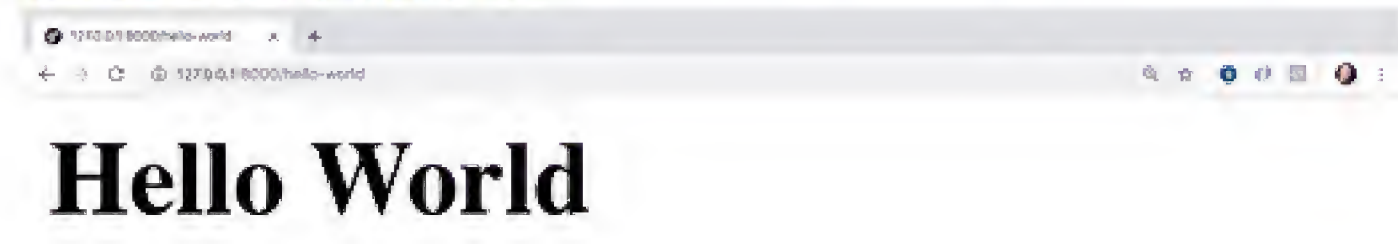
O segundo parâmetro da função `get` do `Route` é o executável para esta rota, que pode ser uma função anônima como vimos na definição da rota principal já existente ou uma string que respeite `Controller@método` e foi como definimos `HelloWorldController@index`.

O Laravel vai chamar o namespace base até o nosso controller e trabalhar em cima da string dada separando o controller do método, criando a instância deste controller e efetuando a chamada do método informado.

Com nossa rota definida, chegamos ao passo 3 e final do nosso roteiro e já podemos iniciar nosso webserver, na raiz do projeto para teste de nosso Hello World. Execute o comand abaixo em seu terminal ou cmd:

```
1 php artisan serve
```


E acesse em seu browser `http://127.0.0.1:8000/hello-world`, onde teremos o resultado abaixo:



A partir de agora iremos desbravar toda estas opções e outros pontos a mais que não foram diretamente mostrados aqui.

Este capítulo serviu para te mostrar um panorama geral que será utilizado no decorrer de sua caminhada com o framework durante o desenvolvimento de suas aplicações mas vamos lá que têm bastante coisa ainda!

Vamos continuar nossa jornada!

O Hello World em nossa tela!

Conclusões

Neste módulo vimos diretamente o processo de expor uma rota para acesso, a execução de nosso controller através do método `index`, bem como, a exibição do resultado em uma view onde mostramos o nosso primeiro Hello World com o framework!

É claro que cada etapa destas carece de um pouco mais de informações e também é claro que cada etapa destas contém diversos detalhes que será necessário que conheçamos para termos um melhor proveito do framework.

Rotas & Controllers

As rotas e controllers são integrantes bem importantes em nossas aplicações Laravel. Neste capítulo vamos conhecer as opções que as rotas nos proporcionam e como podemos trabalhar com controllers conhecendo um pouco além do que vimos no capítulo passado.

Vamos as rotas então!

Rotas

As rotas em nossa aplicação Laravel nos ajudam a termos mais previsibilidade sobre nossas urls. Como nós mapeamos nossas URLs dentro dos arquivos de rotas fica mais fácil termos controle do que será exposto e também fica mais fácil de customizarmos as rotas como queremos.

Dentro do Laravel temos os arquivos de rotas bem separados, que nos ajuda a organizarmos melhor tais rotas que dependendo da aplicação podem se tornar bem grande no quesito de definições dentro do arquivo de rotas em questão.

O Laravel possui os seguintes arquivos de rotas: `web.php`, `api.php`, `channels.php` e `console.php`.

web.php

O arquivo `web.php` conterà as rotas de sua aplicação com as interfaces para o usuário. Todas as rotas que têm esse fim deverão ser definidas neste arquivo.

api.php

Se você for trabalhar com APIs, expondo endpoints para que outras aplicações possam consumir seus recursos você deverá definir suas rotas com este fim no arquivo **api.php**.

channels.php

Se você for trabalhar com eventos de Broadcasting, Notificações e etc suas rotas deverão ser definidas neste arquivo.

console.php

Arquivo para registro de comandos para o console e execução a partir do artisan.

Definição de Rotas

Como vimos no último capítulo, abordei duas formas de definição de rotas em nosso Hello World. Uma era a rota que já existia no arquivo **web.php** e a outra foi nossa definição de rota para nosso Hello World.

Vamos da uma revisada, uma relembrada:

```
1 Route::get('/', function () {
2     return view('welcome');
3 });
```

e

```
1 Route::get('hello-world', 'HelloWorldController@index');
```

Acima temos duas formas de definição para rotas, com respeito ao que será executado. Lembrando que o primeiro parâmetro do método `get` é a rota em questão e o segundo parâmetro será um callable: uma função anônima ou uma string que respeite `Controller@método` que no fim das contas também virará um callable dada a instância do controller e a chamada do método deste controller.

Na rota inicial, que já tínhamos no arquivo `web.php`, vemos a utilização da função anônima e em nossa rota usamos a definição de chamada do controller e seu método diretamente.

Um primeiro ponto que podemos abordar sobre os métodos do Route, como vimos o get até o momento, é que teremos métodos respeitando os verbos http, como:

- GET;
- POST;
- PUT;
- PATCH;
- DELETE;
- OPTIONS;

Podendo utilizá-los da seguinte maneira:

- `Route::get($route, $callback);`
- `Route::post($route, $callback);`
- `Route::put($route, $callback);`
- `Route::patch($route, $callback);`
- `Route::delete($route, $callback);`
- `Route::options($route, $callback);`

Podemos usar os métodos conforme os verbos http mostrados, tendo sempre o primeiro parâmetro, a rota em si, e o segundo parâmetro o callable ou executável para esta rota ao ser acessada.

Route match e Route any

Se precisarmos usar uma rota que responda a determinados tipos de verbos http, podemos usar o método `match` do Route. Como abaixo:

```
1 Route::match(['get', 'post'], 'posts/create', function(){
2     return 'Esta rota bate com o verbo GET e POST';
3 });
```

Caso queira que uma rota responda para todos os verbos ao mesmo tempo, você pode usar o método `any` do Route:

```
1 Route::any('posts', function(){
```

```
2     return 'Esta rota bate com todos os verbos HTTP
3     mencionados anteriormente';
4 });
```

View Routes

Em determinados momentos você vai precisar apenas renderizar determinadas views como resultado do acesso a sua rota. Para isso temos o método `view` do Route que nos permite setar uma rota, primeiro parâmetro, definir uma view, segundo parâmetro, e se preciso podemos passar algum valor para esta view sendo o terceiro parâmetro do método.

Veja como é simples:

```
1 //Exibindo somente a view
2
3 Route::view('/bem-vindo', 'bemvindo');
4
5 //Exibindo a view e mandando parâmetros para ela
6
7 Route::view('/bem-vindo', 'bemvindo', ['name' => 'Nanderson
8 Castro']);
```

Bem simples mesmo, ao acessarmos a rota `bem-vindo` em nosso browser carregaremos a view `bemvindo.blade.php` diretamente como resultado.

Parâmetros dinâmicos

Continuando, vamos conhecer um ponto bem importante sobre rotas que é a possibilidade de informarmos parâmetros dinâmicos. Parâmetros estes que podem servir para identificar determinado recurso como uma postagem em um blog por exemplo.

Veja a rota definida abaixo:


```
1 Route::get('/post/{slug}', function($slug) {
2     return $slug;
3 });
```

Temos a rota `/post/` após isso definimos um parâmetro dinâmico chamado `slug` dentro de chaves como é solicitado pelo componente de rotas. Em nossa função anônima passamos o parâmetro `$slug` que receberá o valor dinâmico e assim, poderemos utilizar ele dentro da nossa função.

Se eu estiver usando um controller e seu método, basta informarmos no método o parâmetro correspondente como informamos na função anônima e utilizarmos tranquilamente.

Com esta rota defininda em nosso arquivo `web.php` e nosso server levantado, podemos acessar em nosso browser a seguinte url:
`http://127.0.0.1:8000/post/teste-parametro-dinamico.`

Resultado:



Como retornamos o parâmetro informado, teremos o valor dinâmico exibido em nossa tela como mostra a imagem acima.

Parâmetros Opcionais

Se você precisar definir parâmetros opcionais para sua rota em questão, basta adicionar a `?` antes do fechamento da última chave do parâmetro na definição da rota. Veja abaixo:

```
1 Route::get('/post/{slug?}', function($slug = null) {
2
3     return !is_null($slug) ? $slug : 'Comportamento sem a
    existência do param slug';
4
5 });
```

Agora nosso parâmetro `slug` é opcional ou seja não será necessário informá-lo na rota e isso nos abre precedente para validarmos ou exibirmos os resultados com base na não existência do parâmetro.

Mais um ponto bem útil nas rotas no Laravel.

Regex em parâmetros

Podemos validar o formato dos parâmetros aceitos em nossas rotas por meio de expressões regulares para um melhor controle. Para isso podemos utilizar o método `where` com este fim:

```
1 Route::get('/user/{id}', function($slug) {
2     return $slug;
3 })
4 ->where(['id' => '[0-9]+']);
```

O método `where` espera um array associativo, sendo a chave o nome do parâmetro e o valor a expressão regular (Regex) a ser validada no parâmetro informado.

Se você tiver mais de um parâmetro dinâmico e quiser informar uma regex para tal, basta ir adicionando no array, respeitando o pensamento chave sendo o parâmetro e valor sendo a expressão regular.

Apelido para rotas

Outro ponto bem importante em nossas rotas são seus apelidos. Mas para que servem? Até agora conhecemos o valor real ou nome real da rota mas podemos chamá-las por meio de seus apelidos também, isso nos ajuda quando precisamos, em um futuro, alterar o nome real das rotas.

Quando fazemos referência aos apelidos, podemos alterar tranquilamente o nome real da rota que o peso desta modificação não será tão impactante assim no quesito negativo. E como utilizar este apelido?

Vamos pegar nossa última rota do parâmetro dinâmico:

```
1 Route::get('/post/{slug}', function($slug) {
2     return $slug;
3 })
4 ->name('post.single');
```

Perceba a adição simples que fiz após o método get antes de fechar com o ;. Chamei o método name que me permite adicionar um apelido para a rota em questão, neste caso agora posso chamar o apelido post_single toda vez que eu precisar usar a rota post/{slug}.

Em um link em nossa view ao invés de usarmos desta maneira:

```
1 <a href="/post/primeiro-post">Primeiro Post</a>
```

Vamos utilizar desta maneira:

```
1 <a href="{route('post.single', ['slug' => 'primeiro-
```

```
post']})}">Primeiro Post</a>
```

Perceba acima que estamos em uma suposta view e utilizamos um método helper do Blade que é o route em nosso atributo href da âncora. O primeiro parâmetro do route é o apelido da rota e se a rota tiver parâmetros dinâmicos, que é o nosso caso, agente informa isso dentro de um array no segundo parâmetro do helper, informando o nome do parâmetro dinâmico e o seu valor.

O método route irá gerar a url correta, informando o parâmetro dinâmico no local correto. Com isso fica mais simples se precisarmos futuramente modificar o nome real da rota por esta questão, de chamarmos a rota pelo apelido ao invés de seu nome real.

Grupo de Rotas & Prefixo

Podemos definir determinadas configurações para um grupo específico de rotas, para conhecermos o poder do group decidir mostrar ele aqui com a definição de um prefixo, método existente no Route também.

Vamos ao código abaixo:

```
1 Route::prefix('posts')->group(function(){
2
3
4     Route::get('/', 'PostController@index')->name('posts.index')
5     ;
6
7     Route::get('/create', 'PostController@create')->name('posts.create');
8
9     Route::post('/save', 'PostController@save')->name('posts.save');
```



```
8
9 });
```

Perceba no set de rotas acima que utilizei inicialmente o método `prefix` e me utilizei do método `group` para definir esse prefixo para um grupo de rotas específico. Esse grupo de rotas é adicionado dentro do método `group` por meio de uma função anônima.

Agora, as rotas dentro do `group` serão prefixadas com o `posts`, ficando desta maneira:

- `/posts/`;
- `/posts/create`;
- `/posts/save`.

Duas rotas acessíveis via GET e uma acessível via POST.

O grupo nos permite esse tipo de configuração, quando precisamos organizar melhor determinadas configurações que se repetirão para mais de um set de rota. Isso melhora até a escrita dos nossos arquivos de rotas e definições.

Grupo de Rotas & Apelidos

Vamos melhorar ainda mais nosso set de rotas do momento passado. Podemos, também, definir um apelido base para um grupo de rotas, então vamos melhorar nosso grupo anterior.

Veja como ficou:

```
1 Route::prefix('posts')->name('posts.')->group(function(){
2
3     Route::get('/', 'PostController@index')->name('index');
4
5     Route::get('/create', 'PostController@create')->name('create
6     ');
```

```
6
7
8     Route::post('/save', 'PostController@save')->name('save');
9 });
```

Perceba que agora isolei a parte `posts`, referente ao apelido das rotas após a definição do prefixo. Agora as rotas deste grupo além de receberem um prefixo, irão receber um apelido base que será concatenado com os apelidos de cada rota do grupo.

Os apelidos ficarão desta forma:

- `posts.index`;
- `posts.create`;
- `posts.save`.

Esses serão os apelidos das rotas gerados, o mesmo que seria anteriormente mas agora com o detalhe de termos organizado e isolado o que era repetido, ou seja, o `posts`..

Grupo de Rotas & Namespaces

O namespace base do Laravel é `App`, e o namespace base dos controllers é `App\Http\Controllers`. Esse namespace é adicionado automaticamente pelo Laravel quando chamamos uma rota referenciando o método de um Controller mas podem existir casos em que você queira criar mais um nível de namespace para um determinado grupo de controllers dentro de sua aplicação.

Por exemplo, podemos ter controllers específicos de um painel administrativo. Suponhamos que temos dentro da pasta controllers uma pasta `Admin` (que também representará mais um nível de namespace) e dentro desta pasta `Admin` temos um `PostController`, um `UserController` ambos referentes ao gerenciamento de posts e usuários de nosso painel administrativo.


```

6
7
Route::post('/save', 'PostController@save')->name('save');
8
9 });

```

Perceba que agora isolei a parte `posts`, referente ao apelido das rotas após a definição do prefixo. Agora as rotas deste grupo além de receberem um prefixo, irão receber um apelido base que será concatenado com os apelidos de cada rota do grupo.

Os apelidos ficarão desta forma:

- `posts.index`;
- `posts.create`;
- `posts.save`.

Esses serão os apelidos das rotas gerados, o mesmo que seria anteriormente mas agora com o detalhe de termos organizado e isolado o que era repetido, ou seja, o `posts`..

Grupo de Rotas & Namespaces

O namespace base do Laravel é `App`, e o namespace base dos controllers é `App\Http\Controllers`. Esse namespace é adicionado automaticamente pelo Laravel quando chamamos uma rota referenciando o método de um Controller mas podem existir casos em que você queira criar mais um nível de namespace para um determinado grupo de controllers dentro de sua aplicação.

Por exemplo, podemos ter controllers específicos de um painel administrativo. Suponhamos que temos dentro da pasta controllers uma pasta `Admin` (que também representará mais um nível de namespace) e dentro desta pasta `Admin` temos um `PostController`, um `UserController` ambos referentes ao gerenciamento de posts e usuários de nosso painel administrativo.

Como podemos referir o namespace `Admin` durante o set de rotas?

Vejamos as duas rotas abaixo:

```

1
Route::get('admin/users/', 'Admin\\UserController@index')->na
me('users.index');
2
3
4
Route::get('admin/posts/', 'Admin\\PostController@index')->na
me('posts.index');

```

Perceba como eu informei o namespace extra, `Admin`, nas duas rotas acima que supostamente levam para a listagem de usuários e posts dentro do nosso admin.

Perceba que temos repetições nas duas rotas, vamos nos focar em organizar o namespace inicialmente.

Organizando o namespace por grupos podemos chegar no resultado abaixo:

```

1 Route::namespace('Admin')->group(function(){
2
3
   Route::get('admin/users/', 'UserController@index')->name('us
   ers.index');
4
5
6
   Route::get('admin/posts/', 'PostController@index')->name('po
   sts.index');
7
8 });

```


Agora isolamos o namespace através do método, do Route, chamado namespace e todas as rotas deste grupo irão receber este namespace.

Podemos ainda organizar os prefixos, perceba a repetição do admin nos nomes reais das rotas. Organizando fica assim:

```
1
Route::namespace('Admin')->prefix('admin')->group(function(){
2
3
Route::get('/users/', 'UserController@index')->name('users.i
ndex');
4
5
6
Route::get('/posts/', 'PostController@index')->name('posts.i
ndex');
7
8 });
```

Agora as coisas começam a ficar mais organizadas.

Podemos organizar e agrupar nossas rotas conforme nossa necessidade, então, sempre que estiver escrevendo suas rotas analise o que pode ser organizado e agrupado com o método group e as opções disponíveis para as configurações em questão.

Controllers

Já tivemos nosso primeiro contato com controllers até este momento. Os controllers são parte importantíssima nesta arquitetura utilizando o Laravel. Relembrando, de forma simples, os controllers são o ponto de delegação entre Model e View, recebendo a requisição e entregando para quem é de direito.

Utilizando o **artisan** podemos automatizar, como já fizemos, a geração

dos nossos controllers. Como por exemplo fizemos no capítulo passado ao executarmos o comando abaixo:

```
1 php artisan make:controller HelloWorldController
```

Agora vamos explorar mais opções dentro deste comando e buscar mais produtividade na execução de nossos projetos.

Controllers como Recurso

Podemos gerar controllers com métodos para cada operação de CRUD e por meio de uma configuração de rota termos também as rotas automáticas para cada um destes métodos.

Este tipo de controller chamamos de controller como recurso. Vamos gerar e entender como são. Em seu terminal na raiz do seu projeto execute o comando abaixo:

```
1 php artisan make:controller UserController --resource
```

Perceba agora que usei o mesmo comando para gerar um novo controller, no caso acima o controller UserController mas adicionei um parâmetro, o `--resource` que criará um controller com os seguintes métodos abaixo:

- index;
- create;
- store;
- show;
- edit;
- update;
- destroy;

Percebe que quando iniciamos o controller como recurso já temos um controller com os métodos acima definidos só no ponto para colocarmos nossas lógicas de inserção, regra de negócio e etc.

Veja o controller gerado abaixo na íntegra:

```

1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6
7 class UserController extends Controller
8 {
9     /**
10      * Display a listing of the resource.
11      *
12      * @return \Illuminate\Http\Response
13      */
14     public function index()
15     {
16         //
17     }
18
19     /**
20      * Show the form for creating a new resource.
21      *
22      * @return \Illuminate\Http\Response
23      */
24     public function create()
25     {
26         //
27     }
28
29     /**
30      * Store a newly created resource in storage.
31      *
32      * @param \Illuminate\Http\Request $request

```

```

33      * @return \Illuminate\Http\Response
34      */
35     public function store(Request $request)
36     {
37         //
38     }
39
40     /**
41      * Display the specified resource.
42      *
43      * @param int $id
44      * @return \Illuminate\Http\Response
45      */
46     public function show($id)
47     {
48         //
49     }
50
51     /**
52      * Show the form for editing the specified resource.
53      *
54      * @param int $id
55      * @return \Illuminate\Http\Response
56      */
57     public function edit($id)
58     {
59         //
60     }
61
62     /**
63      * Update the specified resource in storage.
64      *
65      * @param \Illuminate\Http\Request $request

```



```

66     * @param int $id
67     * @return \Illuminate\Http\Response
68     */
69     public function update(Request $request, $id)
70     {
71         //
72     }
73
74     /**
75      * Remove the specified resource from storage.
76      *
77      * @param int $id
78      * @return \Illuminate\Http\Response
79      */
80     public function destroy($id)
81     {
82         //
83     }
84 }

```

Agora com o controller como recurso gerado, podemos expor rotas para cada um dos métodos acima. Neste caso, para facilitar mais ainda nossas vidas, temos um método dentro do Route chamado de `resource` que já expõe para nós as rotas para cada um destes métodos simplificando ainda mais as coisas.

Em seu arquivo de rotas web adicione a rota abaixo:


```
1 Route::resource('/users', 'UserController');
```

Se nós debugarmos as rotas existentes em nosso projeto até o momento, podemos encontrar, focando no controller como recurso e em suas rotas, o resultado destacado abaixo na imagem.

Para debugarmos as rotas existentes no Laravel, definidas dentro de sua aplicação, basta executar na raiz do seu projeto o comando abaixo:

```
1 php artisan route:list
```

Obtendo a lista de rotas de sua aplicação. Veja o destaque nas rotas criadas pelo método `resource` do Route:



Domain	Method	URI	Name	Action	Middleware
	GET/HEAD	/		Closure	web
	POST	_ignition/execute-solution		Facade\Ignition\ExceptionHandler::exceptionController	Facade\Ignition\ExceptionHandler::exceptionController
	GET/HEAD	_ignition/health-check		Facade\Ignition\HealthCheckController::healthCheckController	Facade\Ignition\HealthCheckController::healthCheckController
	GET/HEAD	_ignition/scripts/{script}		Facade\Ignition\ScriptsController::scriptController	Facade\Ignition\ScriptsController::scriptController
	POST	_ignition/share-report		Facade\Ignition\ShareReportController::shareReportController	Facade\Ignition\ShareReportController::shareReportController
	GET/HEAD	_ignition/styles/{style}		Facade\Ignition\StylesController::styleController	Facade\Ignition\StylesController::styleController
	GET/HEAD	api/user		Closure	api.auth.php
	GET/HEAD	hello-world		App\Http\Controllers\HelloWorldController::index	web
	GET/HEAD	users/{slug}		Closure	web
	GET/HEAD	users	users.index	App\Http\Controllers\UserController@index	web
	POST	users	users.store	App\Http\Controllers\UserController@store	web
	GET/HEAD	users/create	users.create	App\Http\Controllers\UserController@create	web
	GET/HEAD	users/{user}	users.show	App\Http\Controllers\UserController@show	web
	PUT/PATCH	users/{user}	users.update	App\Http\Controllers\UserController@update	web
	DELETE	users/{user}	users.destroy	App\Http\Controllers\UserController@destroy	web
	GET/HEAD	users/{user}/edit	users.edit	App\Http\Controllers\UserController@edit	web

Além de termos gerado o controller com os métodos para serem utilizados dentro de um CRUD completo, temos também a criação das rotas para cada um dos métodos do controller, por meio da chamada de apenas um método, o `resource`. O método `resource` recebe o nome da rota como primeiro parâmetro e o nome do controller, gerado como recurso, no segundo parâmetro, simples assim.

Perceba que na imagem acima temos as rotas, temos também apelidos para estas rotas e ainda temos rotas para cada um dos verbos HTTP.

Entenda abaixo:

- Temos uma rota principal, que serve para listagem de todos os dados, neste caso usuários. Esta rota é `/users` que aponta para o método `index` do `UserController`;
- Temos a rotas `users/create` que aponta para o método `create` do `Usercontroller` que se refere a tela de exibição do form de criação, a rota `users/store` aponta para o método `store` de `UserController` se refere ao salvar de fato os dados vindos do

form de criação;

- Temos ainda a rota para visualização de um dado específico, neste caso, podemos usar tanto a `users/{user}` (aponta para `UserController@show`) ou `users/{user}/edit` (aponta para `UserController@edit`) via GET;
- Para atualização temos a rota `users/{user}` que aponta para o método `update` de `UserController` e também temos o método `destroy` que serve para o remover um dado e têm a rota `users/{user}` e aponta para o método `destroy` do `UserController`. Para atualização verbos PUT ou PATCH e para remoção verbo DELETE do HTTP.

Melhorando a visualização e simplificando:

- rota: `/users/`, verbo: GET, controller@método: `UserController@index`;
- rota: `/users/`, verbo: POST, controller@método: `UserController@store`;
- rota: `/users/create`, verbo: GET, controller@método: `UserController@create`;
- rota: `/users/{user}`, verbo: GET, controller@método: `UserController@show`;
- rota: `/users/{user}`, verbo: PUT ou PATCH, controller@método: `UserController@update`;
- rota: `/users/{user}`, verbo: DELETE, controller@método: `UserController@destroy`;
- rota: `/users/{user}/edit`, verbo: GET, controller@método: `UserController@edit`;

Lembrando que você precisa adicionar sua lógica para cada um dos métodos, entretanto, todas as rotas já estão definidas como vimos acima e quando geramos o controller temos as definições dos métodos também.

Conclusões

Vimos aqui neste capítulo diversas possibilidades sobre as rotas de nossa aplicação e mais opções referentes aos nossos controllers. É claro, que temos mais opções, tanto dentro das rotas quanto na exploração dos controllers.

A cada capítulo vamos conhecendo mais opções para rotas, por exemplo a opção dos middlewares quando conhecermos os conceitos envolvendo esta parte!

No próximo capítulo iremos falar sobre o envio de dados de um formulário para nosso backend e a manipulação das requests e do nosso response.

Até lá!

Formulários, Request & Response

Neste capítulo vamos conhecer um ponto crucial para a criação das nossas aplicações, como manipular informações recebidas em nossas requests e as particularidades do response ou respostas HTTP dentro do Laravel.

Aproveito também e já abordo isso com dados vindos de um formulário onde teremos pequenos pontos importantes a serem abordados neste aspecto.

Então vamos lá!

Iniciando pelo formulário

Para começarmos, vamos criar um formulário para envio dos dados para nosso controller. Para isso crie um arquivo chamado de `create.blade.php` dentro da pasta de views, colocando ele dentro da pasta posts, então crie esta pasta também. O caminho completo até o arquivo ficará assim:

`resources/views/posts/create.blade.php`

Veja o formulário abaixo a ser adicionado no `create.blade.php`:

```
1 <form action="{{route('posts.store')}}" method="post">
2
3     <div class="form-group">
4         <label>Titulo</label>
5         <input type="text" name="title" class="form-
control">
6     </div>
```

```
7
8     <div class="form-group">
9         <label>Descrição</label>
10        <input type="text" name="description" class="form-
control">
11    </div>
12
13    <div class="form-group">
14        <label>Conteúdo</label>
15
16        <textarea name="content" id="" cols="30" rows="10" class="fo
rm-control"></te\
17    </div>
18
19    <div class="form-group">
20        <label>Slug</label>
21        <input type="text" name="slug" class="form-
control">
22    </div>
23
24    <button class="btn btn-lg btn-success">Criar
 Postagem</button>
25 </form>
```

No formulário temos 4 campos diretamente, são eles:

- Título;
- Descrição;
- Conteúdo;
- Slug (futuramente vamos automatizar esta geração);

Obs.: Utilizei a estrutura do form pensando no Twitter Bootstrap, mais a frente no livro vamos linkar esse framework css do Twitter em nossas mas já dando uma pincelada, usei as classes de formulário disponíveis

neste framework são elas: `form-group` e `form-control`. Além das classes para botões: `btn`, `btn-lg`(botão largo) & `btn-success`(que traz a cor verde ao botão).

Nosso formulário trabalhará com envio dos dados via método `POST` e já adicionamos, a `action` do formulário, a chamada de uma rota pelo seu apelido, a `posts.store`. Vamos gerar nosso controller e nossas rotas para que possamos fazer esse form funcionar.

Gere o controller a partir do seu terminal, com o comando abaixo:

```
1 php artisan make:controller Admin/PostController
```

Perceba mais um apredizado acima, como coloquei `Admin/` nosso controller será criado dentro da pasta `Admin` dentro da pasta de controllers. Isso simplifica também nossas gerações.

Pro nosso exemplo não vou gerar como recurso porque nosso foco ainda não é gerarmos um crud e sim conhecermos esse trabalho entre as requisições e o envio de dados.

Uma vez criado o controller, adicione os métodos abaixo:

```
1 public function create()
2 {
3     return view('posts.create');
4 }
```

e também:

```
1 public function store(Request $request)
2 {
3
4 }
```

O primeiro método será para exibição de nossa view criada

anteriormente e o segundo para manipularmos o que for enviado do formulário de criação. Este segundo método veremos mais adiante seu conteúdo.

Até o momento nosso controller fica desta maneira:

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use Illuminate\Http\Request;
6 use App\Http\Controllers\Controller;
7
8 class PostController extends Controller
9 {
10     public function create()
11     {
12         return view('posts.create');
13     }
14
15     public function store(Request $request)
16     {
17
18     }
19 }
```

View criada, controller e métodos definidos vamos adicionar no arquivo de rotas `web.php` nossas definições de rota para este trabalho. Vamos lá!

Veja as rotas adicionadas abaixo:

```
1
Route::prefix('admin')->namespace('Admin')->group(function(){
2
3
Route::prefix('posts')->name('posts.')->group(function(){
```


anteriormente e o segundo para manipularmos o que for enviado do formulário de criação. Este segundo método veremos mais adiante seu conteúdo.

Até o momento nosso controller fica desta maneira:

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use Illuminate\Http\Request;
6 use App\Http\Controllers\Controller;
7
8 class PostController extends Controller
9 {
10     public function create()
11     {
12         return view('posts.create');
13     }
14
15     public function store(Request $request)
16     {
17
18     }
19 }
```

View criada, controller e métodos definidos vamos adicionar no arquivo de rotas web.php nossas definições de rota para este trabalho. Vamos lá!

Veja as rotas adicionadas abaixo:

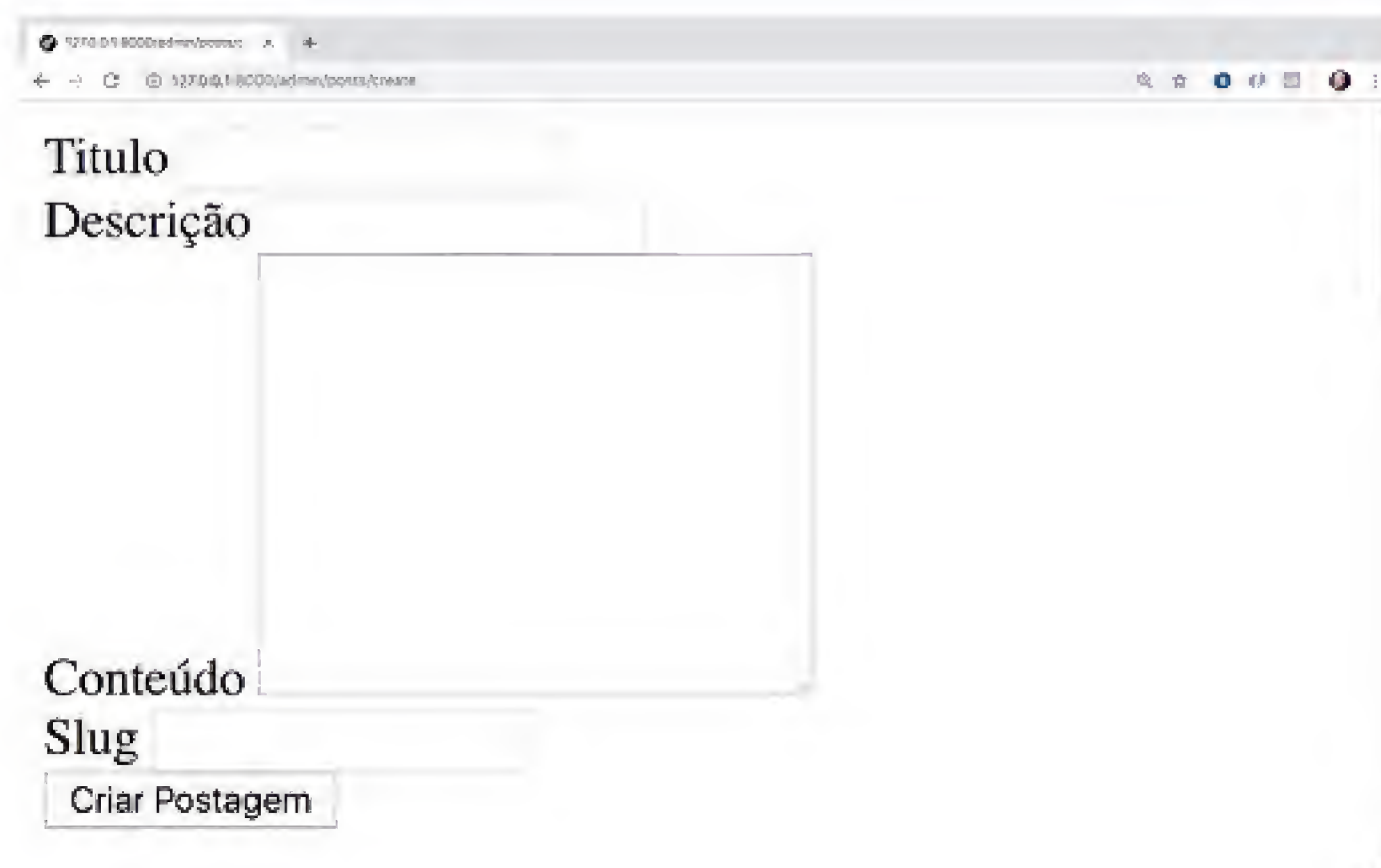
```
1
Route::prefix('admin')->namespace('Admin')->group(function(){
2
3
Route::prefix('posts')->name('posts.')->group(function(){
```

```
4
5
Route::get('/create', 'PostController@create')->name('create'
);
6
7
Route::post('/store', 'PostController@store')->name('store');
8
9     });
10
11 });
```

Perceba acima que isolei o prefixo **admin** e o namespace **Admin** para o grupo de rotas que contém o nosso post. E também isolei o prefixo **posts**, bem como o seu apelido **posts**. para o conjunto de rotas pensadas para este capítulo.

No fim ao acessarmos **/admin/posts/create** veremos nosso formulário, ainda sem estilo, carregado em nossa tela. E também ao enviarmos nosso form ele mandará os dados para nossa rota **/admin/posts/store**.

Com nosso servidor ligado, através do comando `php artisan serve` podemos acessar nosso link **http://127.0.0.1:8000/admin/posts/create** chegando ao resultado da imagem abaixo:



127.0.0.1:8000/admin/posts/create

Titulo

Descrição

Conteúdo

Slug

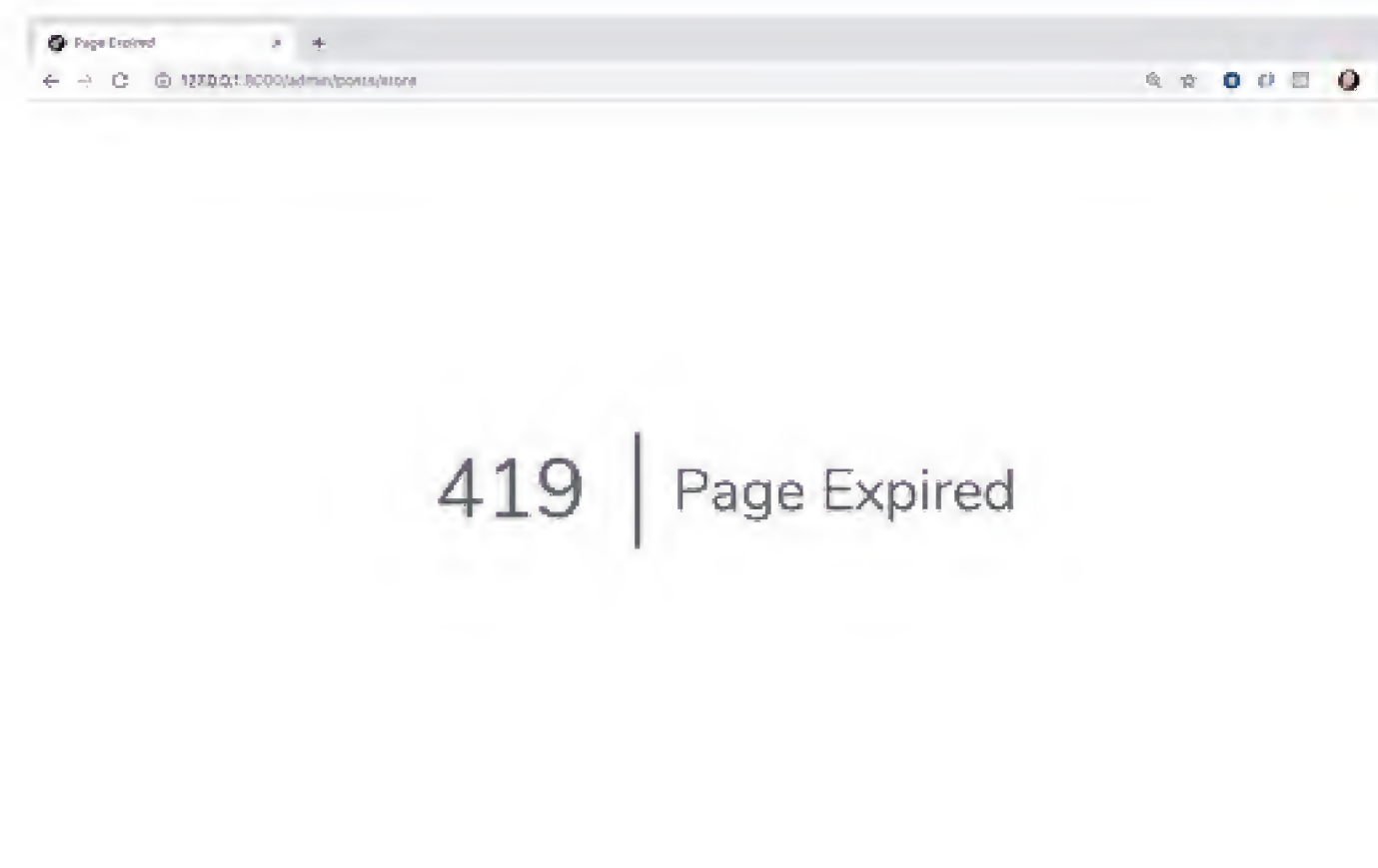
Criar Postagem

Feito isso vamos conhecer o request!

Request, manipulando requisições

Vamos começar este trecho tentando enviar qualquer dado para o nosso backend a partir do formulário. Podemos enviar até o form vazio mesmo, então vamos clicar em **Criar Postagem** em nosso formulário.

Se você recebeu a tela de expirado abaixo, não se preocupe! rrsrrsrrsr!



Por que isso aconteceu? Perceba, pela sua barra de endereços, que ele enviou a requisição para a rota correta: **/admin/posts/store** então porque recebemos uma tela de requisição expirada?

Quando enviamos dados via POST, PUT ou PATCH o Laravel faz uma validação na requisição para evitar que fontes externas enviem dados ou falsifiquem nossa requisição. Esse controle é chamado de **CSRF**.

Obs.: O CSRF não é algo exclusivo do Laravel, é um tópico de segurança e recomendo fortemente a leitura sobre o assunto.

Agora como podemos adicionar esta possibilidade em nosso formulário? Vamos lá então, adicione após a abertura da tag form o seguinte input abaixo:

```
1 <input type="hidden" name="_token" value="
```



```
{{csrf_token()}}">
```

Acima estamos enviando o token csrf, para validar a procedência e envio dos dados do nosso formulário através de nossa requisição. Feito isso, volte para a página do formulário atualize e tente enviar novamente.

Agora você verá tudo branco e a página de expirado já não existe mais, como não temos nada definido no método store lá no controller o resultado será mesmo uma página em branco mas nossa requisição post, vinda do formulário, já bate na execução do método.

O input acima, com o token csrf, pode ser substituído completamente pelo:

```
1 {{csrf_field()}}
```

Que adicionará o input completo como fizemos na mão anteriormente. Ou ainda podemos simplificar mais, utilizando uma diretiva disponível do blade, abaixo:

```
1 @csrf
```

Que também adiciona o input como fizemos anteriormente. O formulário agora fica desta forma:

```
1 <form action="{{route('posts.store')}}" method="post">
2
3     @csrf
4
5     <div class="form-group">
6         <label>Titulo</label>
7         <input type="text" name="title" class="form-
control">
8     </div>
```

```
9
10     <div class="form-group">
11         <label>Descrição</label>
12         <input type="text" name="description" class="form-
control">
13     </div>
14
15     <div class="form-group">
16         <label>Conteúdo</label>
17
18         <textarea name="content" id="" cols="30" rows="10" class="fo
rm-control"></te\
19 xtarea>
20     </div>
21
22     <div class="form-group">
23         <label>Slug</label>
24         <input type="text" name="slug" class="form-
control">
25     </div>
26
27     <button class="btn btn-lg btn-success">Criar
 Postagem</button>
28 </form>
```

Agora que nosso form já está ‘funcionando’ e nossa requisição está chegando no método como podemos manipular os dados dentro do método store?

Manipulando os dados da requisição

Se voltarmos ao método store do PostController, vamos perceber que em sua assinatura temos a definição de um parâmetro.

```
1 public function store(Request $request)
```



```
2 {
3
4 }
```

O parâmetro `$request`, e seu tipo esperado é um objeto `Request` do próprio Laravel. O objeto `Request` vêm do namespace `Illuminate\Http\Request` e ele é automaticamente resolvido pelo container de injeção de dependências do Laravel, mais a frente falaremos sobre esse container.

Com isso, por meio do parâmetro `$request` temos a possibilidade de acesso, das mais variadas formas, aos dados enviados na requisição para nosso método, e em nosso caso, vindos do formulário.

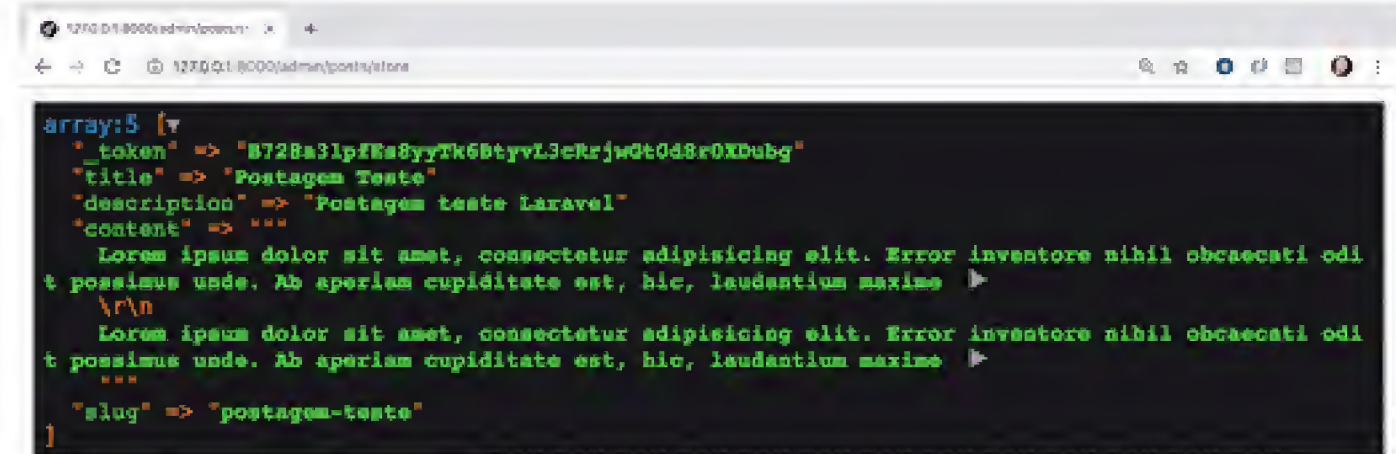
Vamos ver um panorama geral dos dados enviados. Adicione o seguinte trecho dentro do método `store`:

```
1 dd($request->all());
```

Acima temos contato com mais uma função helper do Laravel, o `dd` que simplesmente faz uma dump mais customizado dos dados informados e em seguida joga um `die` travando a continuação da execução do código. Por isso o `dd` ou `dump and die`.

Continuando...

Preencha dados em seu formulário e envie novamente a requisição. Veja o resultado:



O resultado acima são todos os campos vindos do nosso formulário, o resultado do dump é o que temos acima mas os dados estão sendo resgatados por meio do método `all` do objeto `Request` que traz todos os dados enviados por meio desta requisição.

Recuperando valores específicos

Você pode acessar os campos diretamente, por exemplo se eu quiser acessar o título da postagem enviado do formulário eu posso acessar das seguintes maneiras:

```
1 dd($request->get('title'));
```

ou

```
1 dd($request->title);
```


ou ainda

```
1 dd($request->input('title'));
```

Isso vale para cada campo que você envia e que deseja recuperar o valor.

Obs.: Só uma observação, e frisando novamente, estou usando o dd aqui apenas para debug. Mais a frente vamos concluir esta etapa de manipulação, como por exemplo pegar estes valores e salvar no banco de dados.

Verificando a existência de um dado na requisição

Podemos verificar ainda se determinado campo, parâmetro ou input foi informado em nossa requisição. Para isso podemos utilizar o método has do objeto request.

Veja abaixo:

```
1 if($request->has('title')) {
2     dd($request->title);
3 }
```

Com isso podemos realizar determinadas operações para a existência ou não existência dos campos. Se você deseja verificar a existência de vários inputs, você pode utilizar o método hasAny, veja abaixo:

```
1 if($request->hasAny(['title', 'content', 'slug'])) {
2     dd($request->title);
3 }
```

Recuperando campos específicos

Podemos recuperar inputs a nosso gosto caso precisarmos. Por exemplo, se eu quiser recuperar apenas o campo title e o campo slug da nosso

request:

```
1 $request->only(['title', 'slug']);
```

Desta maneira vamos receber um array com os dois campos informados e seus respectivos valores.

Caso você queira ignorar campos específicos também é possível, por exemplo:

```
1 $request->except(['title']);
```

Neste caso receberemos um array com todos os campos e somente o campo title não estará presente neste array. São método bem úteis quando precisamos destes comportamentos.

Trabalhando com Query Strings ou Parâmetros de URL

Parâmetros de url, as famosas query strings, são parâmetros informados em nossa url sempre após a ? e respeitando chave=valor e quando temos mais de um parâmetro são concatenados pelo &.

Por exemplo, como eu poderia recuperar o parâmetro search da seguinte url `http://127.0.0.1:8000?search=teste?`

Para acessarmos seu valor, nós podemos utilizar o método query para recuperar este input exclusivo vindo da url.

Por exemplo:

```
1 $request->query('search');
```

Este método, o query, assim como o get e o input, aceita um segundo parâmetro pro caso da não existência do parâmetro ou input solicitado. Podemos definir um valor default que será carregado caso não tenhamos o input em questão.

Veja abaixo:

```
1 $request->query('search', 'este valor será retornado caso
não tenhamos o parâmetro s\
2 earch na query string');
3
4 //Pro input e pro get
5
6 $request->get('title', 'este valor será retornado caso não
tenhamos o parâmetro titl\
7 e na requisição');
8
9 $request->input('title', 'este valor será retornado caso
não tenhamos o parâmetro ti\
10 tle na requisição');
```

Response, manipulando respostas

Dentro das linguagens Web é importante entendermos o protocolo HTTP e a base dessa arquitetura que é a Web, onde trabalhamos com Requests & Responses e no meio temos os processamentos no backend, seja lá qual for a linguagem.

Manipulamos nossa requisição e de certa forma realizamos processamentos até o momento, agora precisamos conhecer a manipulação das responses ou respostas HTTP.

É claro que abordarei aqui dentro da estrutura do framework mas já deixo uma recomendação de, se caso você não tenha conhecimento básico do http pelo menos, procure mais informações sobre o assunto para entender melhor esta arquitetura da web, isso vai te abrir muito a mente.

Então, como podemos manipular esse tal Response, ou respostas http?

Manipulando as respostas HTTP

Podemos manipular as respostas http em nossos controllers ou funções anônimas em nossas rotas utilizando a função helper response. Veja abaixo uma utilização simples:

```
1 return response('Retornando uma resposta', 200);
```

Acima retornei uma resposta com o conteúdo Retornando uma resposta e o status code http 200, que se refere a status de sucesso.

Podemos definir cabeçalhos HTTP em nossa resposta também. Por exemplo, posso dizer que o tipo da minha resposta é um json da seguinte maneira:

```
1 return response('Retorno do tipo json', 200)
2     ->header('Content-Type', 'application/json');
```

Agora o tipo da resposta, que por default seria do tipo **'text/html'**, será do tipo **'application/json'** por meio da manipulação do cabeçalho http Content-Type e informando o mime-type por meio deste cabeçalho, em nosso caso, colocando o tipo para json: application/json.

Podemos ainda manipular cookies e enviar quantos cabeçalhos forem necessários por meio do objeto response. Por exemplo, se você quiser retornar um cookie em uma determinada resposta, basta utilizar como abaixo:

```
1 return response('Retorno do tipo json', 200)
2     ->cookie('nome_cookie', 'valor_cookie',
'tempo_em_minutos_de_validade_do\
3 _cookie');
```

Redirecionamentos

Dentro desta manipulação do response que têm a ver com o retorno de nossas rotas (quer seja em função anônima quer seja no método de um controller) precisamos abordar, também, sobre redirecionamentos.

Em alguns momentos vamos precisar apenas retornar um redirecionamento pós realização de uma determinada execução. Para isso temos a função helper `redirect`:

```
1 return redirect('/');
```

Acima creio que está bem intuitivo mas não custa comentarmos.

Após um determinado processo posso redirecionar o usuário para uma determinada url, acima redireciono ele para a página inicial do nosso website mas podemos melhorar mais ainda esses redirecionamentos, acima redirecionei ele para uma rota chamando o nome real da rota entretanto eu te disse anteriormente que é melhor trabalharmos com os apelidos da rota ao invés do seu nome real, correto?

Correto, então não se preocupe!

Podemos redirecionar o usuário para a rota desejada por meio do apelido desta rota. Veja abaixo:

```
1 return redirect()->route('home');
```

Chamo a função `redirect` sem parâmetros, com isso terei o retorno do objeto `\Illuminate\Routing\Redirector` e com isso posso ter acesso ao método `route`, que inclusive já utilizamos aqui no livro. Agora basta que eu informe o apelido da rota desejada, e se essa rota possuir parâmetros dinâmicos basta informar em um array no segundo parâmetro do `route` como já vimos anteriormente.

Podemos utilizar também um `redirect` para o estado anterior de uma requisição, isso é perfeito para momentos de erro no processamento de

determinado acesso ou envio de dados. Por exemplo, quando enviamos os dados de um formulário para o backend e temos algum erro de validação nos dados.

Para isso temos a função helper `back`.

Por exemplo:

```
1 return back();
```

Que retornará o usuário para o estado anterior da requisição assim como o botão `back` do browser ou melhor simulando esse comportamento.

Geralmente usamos essa função `back` para momentos de erro de processo ou validações, e nestes casos você precisa voltar o usuário pro momento anterior, caso o usuário tenha mandado dados de um formulário podemos retornar os campos já digitados por ele também como podemos ver abaixo:

```
1 return back()->withInput();
```

Acima além de voltarmos pro estado da requisição anterior, estamos mandando de volta as inputs digitados também. Para manipularmos lá na view, no form e exibir os valores vindos do `withInput` podemos usar a função helper `old` nos inputs do nosso formulário.

Lembra do nosso formulário? Olha como ele fica após adicionarmos essa possibilidade de pegar o valor novamente dos campos digitados anteriormente pelo usuário:

```
1 <form action="{{route('posts.store')}}" method="post">
2
3     @csrf
4
5     <div class="form-group">
```



```

6         <label>Titulo</label>
7         <input type="text" name="title" class="form-
control" value="{{old('title')}}\
8 ">
9     </div>
10
11     <div class="form-group">
12         <label>Descrição</label>
13         <input type="text" name="description" class="form-
control" value="{{old('des\
14 cription')}}">
15     </div>
16
17     <div class="form-group">
18         <label>Conteúdo</label>
19
20         <textarea name="content" id="" cols="30" rows="10" class="fo
rm-control">{{old('content')}}</textarea>
21     </div>
22
23     <div class="form-group">
24         <label>Slug</label>
25         <input type="text" name="slug" class="form-
control" value="{{old('slug')}}">
26     </div>
27
28     <button class="btn btn-lg btn-success">Criar
 Postagem</button>
29 </form>

```

Perceba no textarea e nos atributos value de cada input que temos agora o print do retorno da função helper old onde informados o nome do campo que queremos recuperar o valor digitado anteriormente. Caso

aquele campo em questão não esteja na requisição, adicionado pelo withInput, o campo simplesmente fica em branco e o usuário preenche tranquilamente.

Isso é perfeito para o usuário, que não precisará digitar tudo novamente caso tenhamos algum problema no backend e precisamos retornar pro estado anterior.

Conclusões

Neste capítulo trabalhamos com o envio de dados de um formulário para uma determinada rota e aprendemos também a manipular estes valores além de conhecermos mais detalhes sobre o objeto Request.

Vimos ainda, particularidades sobre as repostas HTTP nos resultados de nossos processos também, além de conhecermos as possibilidades de redirecionamento que estão diretamente atrelados aos responses uma vez que os redirecionamentos também são uma execução pós-processos.

Agora que já entendemos essas manipulações, vamos conhecer como o Laravel trabalha com banco de dados e como podemos manipular a parte do Model fechando o ciclo MVC dentro do nosso livro.

Nos vemos no próximo capítulo.

Database: Migrations, Seeds & Factories

Neste capítulo vamos entrar na área, provavelmente a mais esperada, do banco de dados. Conhecendo como o Laravel nos permite realizar a persistência de dados.

Preciso ressaltar que temos algumas camadas dentro do framework quando tratamos de banco de dados, uma delas é a que vamos abordar neste capítulo, a camada mais baixa, a camada mais estrutural composta por: migrations, seeds & model factories.

Vamos entender o que cada ponto têm de importante aqui.

Começemos pelas migrations.

Migrations

As migrations ou migrações são ferramentas que nos auxiliam no versionamento da estrutura de nossas bases de dados. Funciona basicamente como uma documentação da linha histórica do crescimento da estrutura do banco, criação das tabelas, ligações e etc.

Vale ressaltar que o conceito migrações não é algo do Laravel mas que ele se utiliza para trazer mais essa opção e facilidade para quem está desenvolvendo. Criar tabelas e seus aparatos se torna mais fácil quando realizamos isso do ponto de vista de código que após um comando é traduzido para o banco em questão.

As migrations dentro do nosso projeto podem ser encontradas dentro da pasta database/migrations. Nesta pasta você já vai encontrar alguns arquivos de migração iniciais, como a migração para a tabela de usuários e a para a tabela de reset de senhas, e também temos uma

migration a mais disponível, a da tabela de jobs falhos do sistema de filas do framework.

Vamos dar uma olhada na migration da tabela de usuários e entendermos como é formado um arquivo de migração.

Veja abaixo o arquivo 2014_10_12_000000_create_users_table.php:

```

1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 class CreateUsersTable extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16
17         Schema::create('users', function (Blueprint $table) {
18             $table->bigIncrements('id');
19             $table->string('name');
20             $table->string('email')->unique();
21
22             $table->timestamp('email_verified_at')->nullable();
23             $table->string('password');
24             $table->rememberToken();
25             $table->timestamps();
26         });
27     }
28 }
```



```

25     }
26
27     /**
28      * Reverse the migrations.
29      *
30      * @return void
31      */
32     public function down()
33     {
34         Schema::dropIfExists('users');
35     }
36 }

```

Perceba acima que a classe de migração para a tabela users, inicialmente estende de uma classe base chamada Migration e traz a definição de dois métodos, o método up e o método down.

O método up será executado quando pegarmos essa migração e executarmos em nosso banco de dados. E o método down contém a definição do inverso do método up, no método down nos definimos a remoção do que foi aplicado no método up, isso nos permite voltarmos para estados anteriores pré-execução do último lote de migrações.

Vamos dar uma atenção ao método up:

```

1 public function up()
2 {
3     Schema::create('users', function (Blueprint $table) {
4         $table->bigIncrements('id');
5         $table->string('name');
6         $table->string('email')->unique();
7         $table->timestamp('email_verified_at')->nullable();
8         $table->string('password');
9         $table->rememberToken();

```

```

10         $table->timestamps();
11     });
12 }

```

Temos uma classe base responsável pela definição dos schemas (Schema) da base de dados e é com ela que criamos nossa tabela por meio do método create informando como primeiro parâmetro o nome da tabela e o segundo parâmetro será um callback (ou função anônima) onde definiremos os campos e estrutura da tabela em questão, aqui a tabela users.

Para definir os campos de nossa tabela precisamos do objeto Blueprint que nos permite criarmos os campos e tipos por meio de seus métodos, por isso tipamos o parâmetro \$table, do callback, como Blueprint.

O Blueprint contém métodos para tudo o que é necessário de manipulação de nosso banco, geração e remoção de colunas, os mais variados tipos de dados para as colunas em questão, definição de chaves estrangeiras, criação de índices e muito mais.

Podemos criar nossas chaves primárias e auto-incrementáveis utilizando o método bigIncrements, como temos na linha abaixo:

```
1 $table->bigIncrements('id');
```

Podemos definir campos do tipo string (Varchar), como ocorre abaixo:

```

1 $table->string('name');
2 $table->string('email')->unique();

```

Acima temos a definição de um campo Varchar e por default recebe 255 caracteres, caso queira especificar um tamanho para o campo, basta preencher o segundo parâmetro com o valor inteiro, correspondente ao tamanho do campo desejado.

Perceba também que para o e-mail atribuímos uma definição na coluna, assinalando este campo como unique ou seja evitando a duplicação de linhas com o mesmo email, tornando assim, a partir da base, o usuário único por email.

Podemos definir campos de data e hora por meio do método timestamp, veja abaixo:

```
1 $table->timestamp('email_verified_at')->nullable();
```

O campo email_verified_at que também será nulo, definido pelo método nullable.

Temos também a definição de mais um campo tipo VARCHAR para a coluna password:

```
1 $table->string('password');
```

E por fim temos a chamada do método rememberToken() e também do timestamps(), o que são esses métodos ou que eles fazem?

```
1 $table->rememberToken();
2 $table->timestamps();
```

O método rememberToken irá criar uma coluna chamada de **remember_token**, varchar com tamanho 100 e aceitando o valor nulo. Já o método timestamps irá criar dois campos do tipo timestamp, um chamado de created_at e outro chamada de updated_at ambos representando a data de criação e atualização do dado em questão, o mais interessante é que o Laravel controla os valores destes dois campos automaticamente via Models.

Se o método up define a criação da tabela de users(usuários) o down defini a remoção desta tabela. Veja sua definição:

```
1 public function down()
2 {
3     Schema::dropIfExists('users');
4 }
```

A exclusão ocorre por meio do método dropIfExists do objeto Schema onde informamos a tabela que queremos remover e se ela existir na base, será removida. Isso simplifica bastante pois poderemos voltar um passo anterior se tivermos executado esta migração em algum momento.

Agora como pegamos esse código Orientado a Objetos e jogamos para uma base relacional? É o que vamo ver a seguir.

Executando primeira migração

Se já temos estas migrações disponíveis vamos executá-las em nossa base, epa, espera aí, não temos base ou banco de dados!??

É claro que para executarmos as migrações precisamos está conectados com nossa base de dados em questão. Para isso, na gerenciador de sua escolha crie um banco de dados chamado larave6_ebook_blog e adicione as configurações de acesso em seu arquivo .env na raiz do projeto.

Basta modificar os parâmetros com os valores de sua conexão:

```
1 ;Parâmetros dentro do arquivo .env
2
3 DB_CONNECTION=mysql
4 DB_HOST=127.0.0.1
5 DB_PORT=3306
6 DB_DATABASE=larave6_ebook_blog
7 DB_USERNAME=root
8 DB_PASSWORD=
```


Caso você esteja utilizando outro banco de dados que não mysql será necessário alterar o drive na variável DB_CONNECTION.

Para testarmos se nossa conexão ocorreu com sucesso, no cenário em que estamos no momento, vamos ao nosso terminal e na raiz do projeto vamos executar o comando abaixo:

```
1 php artisan migrate:install
```

Se tudo ocorrer bem como mostra o resultado abaixo, sua conexão está correta e setada com sucesso:

```
blog: php artisan migrate:install
Migration table created successfully.
blog: █
```

Agora o que este comando que executamos acabou de fazer? Bem simples, ele apenas criou a tabela de controle de migrações executadas na base. Se você acessar sua base verá que existe lá uma tabela chamada de migrations que registra o nome da migração executada e o lote em que esta migração foi executada.

Ainda não executamos a execução das migrações existentes no projeto até o momento, então, como realizamos esta execução?

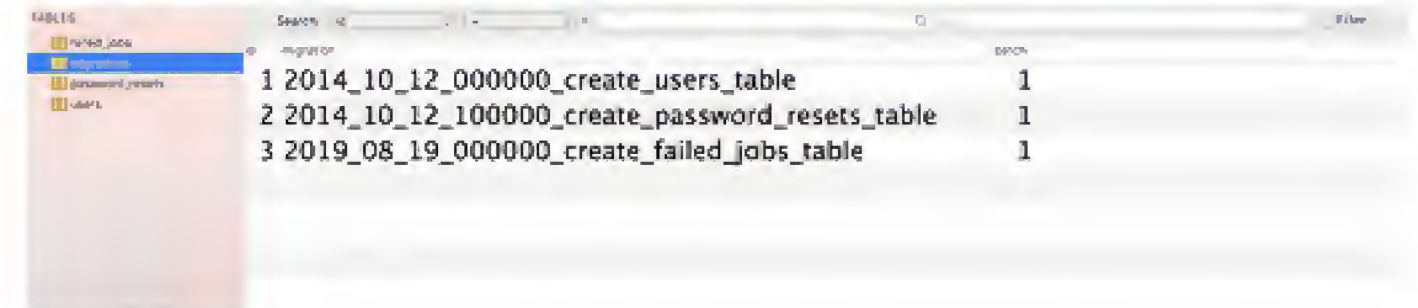
Para rodarmos e executarmos os arquivos de migração existentes é necessário executar o comando abaixo:

```
1 php artisan migrate
```

Resultado:

```
blog: php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.14 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.08 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.07 seconds)
blog: █
```

Veja que agora cada arquivo de migração existente foi executado em nossa base e já estão registrados na tabela migrations com o lote (coluna batch) como 1, primeiro lote de execução:



id	migration	batch
1	2014_10_12_000000_create_users_table	1
2	2014_10_12_100000_create_password_resets_table	1
3	2019_08_19_000000_create_failed_jobs_table	1

E é claro as tabelas também foram criadas e estão em nosso banco agora.

Mas como posso criar minhas migrações para tabelas do meu projeto também? Certo! Vamos fazer isso agora!

Criando Nossas Migrações

Primeiro passo é irmos ao nosso terminal e executarmos o comando para geração de nosso arquivo de migração:

```
1 php artisan make:migration create_table_posts
--create=posts
```



```
blog: php artisan make:migration create_table_post --create=posts
Created Migration: 2019_09_23_095103_create_table_post
blog: █
```

O comando acima criará nosso primeiro arquivo de migração dentro da pasta de migrações, chamado de `2019_09_23_095103_create_table_post`, o nome do arquivo de migração respeita a data de criação mais o timestamp e o nome escolhido, em nosso caso: `create_table_posts`. Essa definição da data e timestamp permite o Laravel organizar a ordem das migrações.

O parâmetro `--create=posts` adicionará para nós o código da classe `Schema` e o método `create` como podemos ver no conteúdo do arquivo gerado abaixo:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateTablePost extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16
17         Schema::create('posts', function (Blueprint $table) {
```

```
18             $table->timestamps();
19         });
20     }
21
22     /**
23      * Reverse the migrations.
24      *
25      * @return void
26      */
27     public function down()
28     {
29         Schema::dropIfExists('posts');
30     }
31 }
```

Por termos utilizado o parâmetro `--create` além das definições do métodos `up` e `down` foram adicionados seus conteúdos com alguns detalhes de campos iniciais no `up`, a definição do campo de auto incremento e a definição dos campos de criação e atualização dos registros.

O `down` já trouxe o movimento contrário, neste caso a remoção da tabela `posts`.

Agora vamos as nossas adições, a adição dos nossos campos para nossa tabela de `posts`. Criaremos os seguintes campos:

- `title`: string 255;
- `description`: string 255;
- `content`: text;
- `slug`: string 255;
- `is_active`: boolean;

Agora como podemos representar estes campos acima dentro do nosso arquivo? Vamos lá, após a definição do `bigIncrements` defina o código

abaixo:

```
1 $table->string('title');
2 $table->string('description');
3 $table->text('content');
4 $table->string('slug');
5 $table->boolean('is_active');
```

Simples, acima realizamos as definições dos nossos campos. Agora estamos aptos a executar esta migração em nosso banco de dados, para isso vamos ao nosso terminal executar o comando que já conhecemos.

Veja abaixo:

```
1 php artisan migrate
```

Ao executarmos o comando acima novamente, o Laravel só executará as migrações que ainda não foram executadas. Em nosso caso, e no momento, a única que não foi executada foi a que geramos acima. Por isso teremos o resultado abaixo:

```
blog: php artisan migrate
Migrating: 2019_09_23_095103_create_table_post
Migrated: 2019_09_23_095103_create_table_post (0.06 seconds)
blog: █
```

Relacionamentos via Migrations

Agora que temos nossa tabela posts criada, vamos mapear nosso primeiro relacionamento entre posts e usuários caracterizando assim a relação de posts e autor. O relacionamento aqui que irei mapear será de 1:N (Um para Muitos) onde 1 autor(usuário) poderá ter N(vários) posts e 1 post poderá ter ou pertencer a apenas um 1 autor/usuário.

Como estamos definindo nossa base via migrations vamos aprender aqui

a definir este relacionamento e de quebra saber como alterar uma tabela já existente por meio de migrations, neste caso alterar posts para adicionarmos a referência para user e ainda criar nossa chave estrangeira.

Para isso execute o comando abaixo:

```
1 php artisan make:migration
alter_table_posts_add_column_user_id --table=posts
```

Perceba que o comando continua o mesmo, eu apenas criei outro arquivo e aqui temos a chamada de um novo parâmetro, quando queremos criar uma tabela e suas definições nós utilizamos o método, do objeto Schemam, chamado de create como vimos no trecho passado. Agora que eu quero alterar uma tabela já existente eu preciso utilizar o método table por isso chamei o parâmetro --table e o nome da tabela posts, este parâmetro vai gerar o conteúdo do arquivo conforme podemos ver abaixo:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class AlterTablePostsAddColumnUserId extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
```



```

16 Schema::table('posts', function (Blueprint $table) {
17     //
18     });
19 }
20
21 /**
22  * Reverse the migrations.
23  *
24  * @return void
25  */
26 public function down()
27 {
28     Schema::table('posts', function (Blueprint $table) {
29         //
30         });
31     }
32 }

```

Ele já traz os conteúdos do método up e do método down chamando o método table, o que nos resta é só adicionarmos nossas modificações para a tabela desejada. O nome da tabela vai no primeiro parâmetro do método table e o segundo, mesmo pensamento do create, vai as definições para a tabela na função anônima ou callback.

Como vamos adicionar a referência para user, irei criar um campo user_id respeitando as mesmas configurações do campo id na tabela de users para que nossa ligação na chave estrangeira possa ser satisfeita.

Precirei criar um campo do tipo bigInteger e unsigned para isso adicione no método up a definição abaixo:

```
1 $table->unsignedBigInteger('user_id')->after('id');
```

A definição já satisfaz o tipo e configurações esperadas para esta referência, e aqui temos mais um conhecimento. Como estou alterando a tabela posts pós criação, posso informar em que posição quero que o campo novo seja adicionado, neste caso, depois do campo id de posts e isso é realizado pelo método after(em pt depois).

Agora, ainda no método up, precisamos criar nossa chave estrangeira para a coluna user_id referenciando a coluna id na tabela users. Adicione o trecho abaixo, após a definição anterior.

```
1 $table->foreign('user_id')->references('id')->on('users');
```

Acima assinalei a criação da chave estrangeira com o método foreign informando o nome da coluna, no caso user_id e informei o campo e a tabela remota para esta referência, neste caso o campo id por meio do método references e a tabela users por meio do método on. Ou seja se fossemos traduzir: crie uma chave estrangeira para o campo user_id de posts que faz referência para o id lá na tabela users.

O nosso método down, por ser o reverso do up, conterà a remoção da chave estrangeira e também da coluna user_id. Então, adicione as duas definições abaixo:

```
1 $table->dropForeign('posts_user_id_foreign');
2 $table->dropColumn('user_id');
```

Quando o Laravel, por meio das migrations, cria a chave estrangeira ela recebe o nome respeitando a estrutura abaixo:

```
1 tabela_coluna_foreign
```

Por isso no down estou apagando a chave estrangeira, por meio do método dropForeign e informando a string posts_user_id_foreign e logo após, por meio do método dropColumn removo a coluna user_id completando assim o reverso do que é executado no método up.

Agora para que isso seja executado em nossa base, basta irmos ao nosso terminal e executarmos o comando abaixo:

```
1 php artisan migrate
```

Comando que já conhecemos e que vai executar a última migração criada, porque ainda não temos registro dela, na tabela de migrations. Veja o resultado:

```
blog: php artisan migrate
Migrating: 2019_09_23_135618_alter_table_posts_add_column_user_id
Migrated: 2019_09_23_135618_alter_table_posts_add_column_user_id (0.13 seconds)
blog: █
```

Agora já temos nossa tabela para salvar as postagens e também nossa referência para criação do autor da postagem em questão. Sobre migrations, por enquanto, ficaremos por aqui.

É claro que ainda veremos bastante elas aqui no livro mas por hora já tivemos um excelente conhecimento a respeito de sua utilização. Agora vamos conhecer mais dois caras que nos auxiliam no momento do nosso desenvolvimento e banco de dados.

Os **seeds** e as **factories**.

Seeds

Quando estamos em nosso ambiente de desenvolvimento nós precisaremos criar dados no banco para teste de nossas lógicas, os seeds nos permitem este processo. Podemos por meio das seeds alimentar nosso banco de dados com informações para que possamos testar nossos CRUDs por exemplo.

Você pode encontrar os seeds do sistema dentro da pasta `database/seeds`, dentro desta pasta você vai encontrar o arquivo `DatabaseSeeder.php` que é o arquivo principal que executa todas as

outras seeds que tivermos nesta pasta.

Perceba que temos seu conteúdo mostrado abaixo:

```
1 <?php
2
3 use Illuminate\Database\Seeder;
4
5 class DatabaseSeeder extends Seeder
6 {
7     /**
8      * Seed the application's database.
9      *
10     * @return void
11     */
12     public function run()
13     {
14         // $this->call(UsersTableSeeder::class);
15     }
16 }
```

Perceba a linha comentada do método `$this->call`, que contém a chamada para um arquivo de seed que ainda não existe mas já nos mostra como podemos registrar os arquivos de seed para serem executados, ou seja, por meio do método `call` e informando a classe de seed em questão.

Vamos criar dois arquivos de seed para este momento, primeiro o arquivo que já temos referenciado acima, o `UsersTableSeeder` e também vamos criar o `PostsTableSeeder`.

Ta! Mas como podemos criar isso? Eu te mostro, é bem simples!

Exeute os comandos abaixo, primeiro executando a geração de um, depois do outro:


```
1 php artisan make:seeder UsersTableSeeder
```

Após a execução acima, execute a criação do outro arquivo seeder:

```
1 php artisan make:seeder PostsTableSeeder
```

Veja o resultado:

```
blog: php artisan make:seeder UsersTableSeeder
Seeder created successfully.
blog: php artisan make:seeder PostsTableSeeder
Seeder created successfully.
blog: █
```

Após a criação, veja que temos dois arquivos criados dentro da pasta de seeds, o UsersTableSeeder.php e também o PostsTableSeeder.php.

Agora o que faremos com eles?

Podemos utilizar ambos os arquivos de seed para gerarmos 1 usuário e 1 postagem para nossa base. Como fazer isso?

Abra o arquivo UsersTableSeeder.php e adicione o código abaixo no método run:

```
1 \DB::table('users')->insert([
2     'name'      => 'Primeiro Usuário',
3     'email'     => 'email@email.com',
4     'password' => bcrypt('secret')
5 ]);
```

Acima temos o primeiro contato com o objeto DB que nos permite realizarmos a execução de queries SQL no mais baixo nível em relação a parte do ORM que veremos mais a frente. Então informo a tabela que quero realizar a inserção do dado, por meio do método table e logo após, aninhando, chamo o método insert informando um array

respeitando as colunas que quero adicionar valor e seus valores em questão.

Perceba que no campo de senha do usuário utilizo o helper **bcrypt** para encryptar a senha do nosso usuário.

Após isso adicione o conteúdo do método run do PostsTableSeeder:

```
1 \DB::table('posts')->insert([
2     'title'      => 'Primeira Postagem',
3     'description' => 'Postagem teste com seeds',
4     'content'    => 'Conteúdo da postagem',
5     'is_active'  => 1,
6     'slug'       => 'primeira-postagem',
7     'user_id'    => 1
8 ]);
```

Acima temos o mesmo pensamento com a diferença que estamos lidando com posts e respeitando o nome da tabela e os campos. Perceba também que referenciei o user_id como 1, isso é pouco chato de se fazer assim mas neste caso se encaixa tranquilamente, uma vez que vamos executar a seed de users na ordem antes de posts onde teremos o usuário com id 1 para satisfazer com o user_id da postagem definida acima.

Agora vamos voltar lá no DatabaseSeeder e descomentar a linha que temos definida e adicionar mais uma para a chamada do PostsTableSeeder.

O conteúdo do arquivo DatabaseSeeder ficará assim, veja ele todo na íntegra abaixo:

```
1 <?php
2
3 use Illuminate\Database\Seeder;
```



```

4
5 class DatabaseSeeder extends Seeder
6 {
7     /**
8      * Seed the application's database.
9      *
10     * @return void
11     */
12     public function run()
13     {
14         $this->call(UsersTableSeeder::class);
15         $this->call(PostsTableSeeder::class);
16     }
17 }

```

Temos a chamada descomentada do UsersTableSeeder e adicionamos a chamada para o PostsTableSeeder. Este passo feito, vamos ao terminal e conhecer mais um comando. Desta vez para execução dos nossos seeds.

Em seu terminal e na raiz do projeto execute o comando abaixo:

```
1 php artisan db:seed
```

Veja o resultado:

```

blog: php artisan db:seed
Seeding: UsersTableSeeder
Seeding: PostsTableSeeder
Database seeding completed successfully.
blog: █

```

Se você for ao seu banco e consultar as tabelas verá que tens os dados lá como definimos nas classes de seed.

Se precisarmos de mais dados, como por exemplo, inserir 30 posts de primeira por meio dos seeds teríamos um trabalho animal mas este trabalho se simplifica por meio do que chamamos de **Model Factories**.

Então vamos crescer mais dentro do Laravel conhecendo mais este conceito/ferramenta que nos auxilia neste camada/etapa do desenvolvimento.

Vamos continuando...

Factories

Quando precisamos realizar a geração de muitos dados de forma mais automatizada podemos utilizar as **Factories** ou **Model Factories**.

Vamos conhece-las por meio da execução.

Aqui de certa forma teremos algum contato sobre models mas não vou me ater a elas (Models) no momento, pois o próximo capítulo será sobre esta camada e o tratamento com banco de dados. Como precisamos de models para geração automatizada de dados para nossa aplicação vou começar primeiramente com o model que já temos em nossa aplicação, o model Users.

As factories definem as regras, com base no model escolhido e conforme a quantidade especificada, para criação de dados fictícios em nossas tabelas. No fim das contas, são chamados de factories ou fábricas por serem um esboço para criação de dados e junto com os seeds realizamos as gerações esperadas. Você pode encontrar os arquivos de factories na pasta database/factories.

Nesta pasta já temos um arquivo de factory, o UserFactory.php. Veja o conteúdo dele abaixo:

```

1 <?php
2

```



```

3 /** @var \Illuminate\Database\Eloquent\Factory $factory */
4 use App\User;
5 use Faker\Generator as Faker;
6 use Illuminate\Support\Str;
7
8 /*
9
10 |-----
11 | Model Factories
12 |-----
13 | This directory should contain each of the model factory
14 | definitions for
15 | your application. Factories provide a convenient way to
16 | generate new
17 | model instances for testing / seeding your application's
18 | database.
19 |
20 | */
21 $factory->define(User::class, function (Faker $faker) {
22     return [
23         'name' => $faker->name,
24         'email' => $faker->unique()->safeEmail,
25         'email_verified_at' => now(),
26
27         'password' => '$2y$10$92IXUNpkj00r0Q5byMi.Ye4oKoEa3Ro9llC/.o
28         g/at2.uhewG/igi'\
29         , // password
30
31         'remember_token' => Str::random(10),

```

```

27 ];
28 });

```

Perceba acima que temos a definição para criação de usuários, um código diretamente definido sem uma casca(objeto ou coisa assim). Um ponto importante aqui nesta definição é a utilização do pacote Faker que nos permite gerar dados aleatórios dos mais variados, como nomes fakes, emails, conteúdos de textos, valores decimais e muito mais.

Vale ressaltar que o pacote Faker é uma lib a parte mas que o Laravel se utiliza para facilitar esse esboço para criação de dados fictícios para teste de nossas aplicações.

Veja que a estrutura quase que se assemelha ao que fizemos no seed no que se refere a dizermos as colunas referentes a tabela, aqui user, e para o que colocamos na mão do dado a ser salvo, aqui usamos o Faker para gerarmos nomes fakes, emails fakes únicos.

Agora, lembrando, aqui temos um esboço mas e como executar esta geração?

Vamos ver a seguir.

Model Factories com Seeds

Como comentado vamos executar nossa inserção de dados utilizando o dado esboçado pela factory. Como o título do trecho diz, precisamos combinar a geração dos dados, utilizando a factory, junto com a execução via seeds. Agora, vamos alterar nosso UsersTableSeeder para utilizar as factories.

Comente o conteúdo do método run do UsersTableSeeder e adicione a chamada abaixo:

```

1 factory(\App\User::class, 10)->create();

```



```
27     ];
28 });
```

Perceba acima que temos a definição para criação de usuários, um código diretamente definido sem uma casca(objeto ou coisa assim). Um ponto importante aqui nesta definição é a utilização do pacote Faker que nos permite gerar dados aleatórios dos mais variados, como nomes fakes, emails, conteúdos de textos, valores decimais e muito mais.

Vale ressaltar que o pacote Faker é uma lib a parte mas que o Laravel se utiliza para facilitar esse esboço para criação de dados fictícios para teste de nossas aplicações.

Veja que a estrutura quase que se assemelha ao que fizemos no seed no que se refere a dizermos as colunas referentes a tabela, aqui user, e para o que colocamos na mão do dado a ser salvo, aqui usamos o Faker para gerarmos nomes fakes, emails fakes únicos.

Agora, lembrando, aqui temos um esboço mas e como executar esta geração?

Vamos ver a seguir.

Model Factories com Seeds

Como comentado vamos executar nossa inserção de dados utilizando o dado esboçado pela factory. Como o título do trecho diz, precisamos combinar a geração dos dados, utilizando a factory, junto com a execução via seeds. Agora, vamos alterar nosso UsersTableSeeder para utilizar as factories.

Comente o conteúdo do método run do UsersTableSeeder e adicione a chamada abaixo:

```
1 factory(\App\User::class, 10)->create();
```

Ao invés de usarmos o objeto DB, como fizemos, chamaremos o helper factory informando o model para o qual queremos gerar os dados e passando um segundo parâmetro onde informamos a quantidade de dados que queremos inserir em uma execução. Com isso chamamos o método create para realizar a execução dos dados e criação de 10 usuários aleatórios.

Depois desta definição na classe UsersTableSeeder vamos ao terminal e vamos rodar somente o seed para a classe de seed de usuário. Para isto execute o comando abaixo:

```
1 php artisan db:seed --class=UsersTableSeeder
```

Resultado:

```
blog: php artisan db:seed --class=UsersTableSeeder
Database seeding completed successfully.
blog: █
```

Acima conhecemos mais uma possibilidade dos seeds, podemos executar seeds especificando a classe que queremos executar, em nosso caso apenas o conteúdo do UsersTableSeeder informando o parâmetro --class e o nome da classe.

Se você for ao seu banco, além do dado que você inseriu no momento anterior com o seed você verá mais 10 registros que foram gerados neste último comando por meio da execução da factory via execução dos seeds. Perceba que isso simplifica bastante nossas vidas, pois se quisermos mais dados fakes basta definirmos a quantidade no parâmetro do helper factory ou até mesmo ficar executando os seeds várias vezes.

Nossa primeira factory

Vamos fazer um trabalho agora, criando uma factory do zero até sua execução por meio do ou em combinação com os seeds. Para criarmos

nossa primeira factory, acesse seu terminal e execute o comando abaixo:

```
1 php artisan make:factory PostFactory
```

Você poderá encontrar a factory criada na pasta em questão. Abra ela e vamos as definições para esta factory. Inicialmente temos o código na ítegra abaixo sem alterações:

```
1 <?php
2
3 /** @var \Illuminate\Database\Eloquent\Factory $factory */
4
5 use App\Model;
6 use Faker\Generator as Faker;
7
8 $factory->define(Model::class, function (Faker $faker) {
9     return [
10         //
11     ];
12 });
```

Primeira coisa que precisamos definir é o model Post entretanto não temos este model ainda, isso não é problema pro momento. Vamos ao terminal e vamos gerar nosso primeiro Model:

```
1 php artisan make:model Post
```

```
blog: php artisan make:model Post
Model created successfully.
blog: █
```

Após a execução do comando acima, você poderá encontrar seu primeiro model gerado na pasta app. Como comentei estamos tendo

aqui neste trecho nossos primeiros contatos com os models mas não vamos adentrar ainda neles, vamos nos focar em sua relação com as factories para geração de dados fakes.

Agora que temos nosso model Post gerado, podemos alterar a definição da factory, que esta assim:

```
1 $factory->define(Model::class, ...
```

para

```
1 $factory->define(\App\Post::class, ...
```

Agora precisamos definir dentro da função anônima, por meio do Faker, nossos dados falsos para futuras gerações de postagens para testes de nossa aplicação. Vamos as definições abaixo:

```
1 return [
2     'title'      => $faker->words(4, true),
3     'description' => $faker->sentence,
4     'content'    => $faker->paragraphs(9, true),
5     'slug'       => $faker->slug,
6     'is_active'  => $faker->boolean,
7     'user_id'    => rand(1, 10)
8 ];
```

Acima temos o array de retorno do nosso callback, vamos entender cada definição fake acima:

- **words(4, true):** estou definindo 4 palavras pro título e essa geração têm que ser em texto por isto o true como segundo parâmetro;
- **setence:** gero aqui uma setença de texto para a descrição do post;
- **paragraphs(9, true):** este é mais intuitivo, espero gerar 9 parágrafos como texto, por isso o segundo parâmetro como true,

caso false ele retorna um array com os parágrafos, o mesmo vale pro segundo parâmetro do word acima.

- **slug**: temos definições específicas para slug;
- **boolean**: e definições aleatórias para true ou false;
- por fim, no `user_id` usei o **rand** do PHP para pegar um inteiro aleatório entre 1 e 10 pensando nos 10 usuários gerados com faker anteriormente.

Obs.: Sobre essa referência ao `user_id` podemos realizar criações mais organizadas e menos adivinhações mas ainda não chegamos neste conhecimento. Não se preocupe que terei o cuidado de mencionar o conhecimento quando chegarmos nele e inclusive vamos voltar e melhorar nossas factories no momento de suas gerações.

Com isto definido, veja o conteúdo na íntegra do PostFactory antes de chamarmos em nosso PostsTableSeeder:

```
1 <?php
2
3 /** @var \Illuminate\Database\Eloquent\Factory $factory */
4
5 use App\Model;
6 use Faker\Generator as Faker;
7
8 $factory->define(\App
9 \Post::class, function (Faker $faker) {
10     return [
11         'title'      => $faker->words(4, true),
12         'description' => $faker->sentence,
13         'content'    => $faker->paragraphs(9, true),
14         'slug'        => $faker->slug,
15         'is_active'   => $faker->boolean,
16         'user_id'     => rand(1, 10)
17     ];
```

```
17 }));
```

Agora vamos adicionar a chamada para execução dentro do nosso PostsTableSeeder, comente o DB insert usado anteriormente e adicione ao método run o trecho abaixo:

```
1 factory(\App\Post::class, 30)->create();
```

Agora quando rodarmos nosso seed para posts, vamos gerar 30 postagens fakes. Vamos executar então, no terminal.

Execute o comando abaixo:

```
1 php artisan db:seed --class=PostsTableSeeder
```

```
blog: php artisan db:seed --class=PostsTableSeeder
Database seeding completed successfully.
blog: █
```

Agora temos, 31 postagens por conta das 30 usando factories e da 1 utilizando o objeto DB. É muito simples gerarmos dados fakes usando essa combinação, definição por meio das factories e execução das factories por meio dos seeds. Isso facilitará muito os testes de sua aplicação no que diz respeito do trabalho com dados.

Antes de concluirmos este capítulo, quero mostrar mais algumas coisas sobre migrations, alguns passos para realizarmos o caminho reverso. Execuções pro caso de queremos limpar nossas tabelas por meio das migrations, limpar e re-criar usando ainda os seeds junto, esses pontos que vão nos auxiliar mais ainda nesta etapa de desenvolvimento.

Vamos lá!

Migrations! Revertendo coisas!

Agora que já conhecemos como criar tabelas e como alimentá-las com dados falsos para nosso desenvolvimento vamos entender como podemos reverter as coisas quando tratamos das migrações e suas criações.

Primeramente caso precisemos voltar o último lote de migrações executados, podemos usar o comando abaixo:

```
1 php artisan migrate:rollback
```

O comando acima, como comentado, desfazá tudo que as migrações do último lote de execução fez. Em nosso caso, foi a adição do autor do post. Você pode voltar também as execução por meio de passos, informando o parâmetro `--step`:

```
1 php artisan migrate:rollback --step=2
```

Neste caso acima, ele desfazá as duas migrações mais recentes, em nosso caso a criação da tabela `posts` e ainda a remoção da tabela de `failed_jobs`. Tabela `posts` criada por nós e `failed_jobs` que já vêm definido nas migrações do projeto.

Podemos ainda realizar operações em cima de todas as migrações, desfazendo-as, de uma vez só com o comando abaixo:

```
1 php artisan migrate:reset
```

O comando acima desfazá todas as migrações executadas em sua base, deixando apenas a tabela `migrations` mas vazia. Caso queiramos resetar mas executar as migrações novamente e ao mesmo tempo, podemos executar o comando abaixo:

```
1 php artisan migrate:refresh
```

Ou ainda executar tudo do zero novamente com o comando `fresh`, que

apaga todas as tabelas e executa as migrations novamente:

```
1 php artisan migrate:fresh
```

Este comando acima, o `migrate:fresh` executa um `drop table` na base para cada tabela. Mesmo para tabelas que não foram executadas via migration, então, fique em alerta com este comando.

Refresh com execução de seeds

Caso queira voltar todos os passos executados nas migrations e na re-execução, executar as seeds podemos informar o parâmetro `--seed` junto com o comando `refresh`. Veja abaixo:

```
1 php artisan migrate:refresh --seed
```

Com isso teremos a execução reversa das migrações, sua re-execução e ainda a execução das seeds no banco de dados. Veja o resultado do comando abaixo:

```
blog: php artisan migrate:refresh --seed
Rolling back: 2019_09_23_135618_alter_table_posts_add_column_user_id
Rolled back: 2019_09_23_135618_alter_table_posts_add_column_user_id (0.04 seconds)
Rolling back: 2019_09_23_095103_create_table_post
Rolled back: 2019_09_23_095103_create_table_post (0 seconds)
Rolling back: 2019_08_19_000000_create_failed_jobs_table
Rolled back: 2019_08_19_000000_create_failed_jobs_table (0 seconds)
Rolling back: 2014_10_12_100000_create_password_resets_table
Rolled back: 2014_10_12_100000_create_password_resets_table (0.01 seconds)
Rolling back: 2014_10_12_000000_create_users_table
Rolled back: 2014_10_12_000000_create_users_table (0.01 seconds)
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.04 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.04 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.02 seconds)
Migrating: 2019_09_23_095103_create_table_post
Migrated: 2019_09_23_095103_create_table_post (0.02 seconds)
Migrating: 2019_09_23_135618_alter_table_posts_add_column_user_id
Migrated: 2019_09_23_135618_alter_table_posts_add_column_user_id (0.05 seconds)
Seeding: UsersTableSeeder
Seeding: PostsTableSeeder
Database seeding completed successfully.
blog: █
```


Conclusões

Bom, este capítulo foi bem denso e puxado. Recomendo fortemente que você execute bastante os comandos aprendidos aqui e teste cada nuances dos comandos para reverter coisas no banco via migrations.

No próximo capítulo vamos subir o nível de utilização da camada de banco tratando agora diretamente com as Models e suas representações. Então até lá!

Eloquent, trabalhando com Models

Continuando nosso trabalho com a camada de dados e persistência, vamos subir o nível conhecendo a camada dos models e como podemos trabalhar buscas, inserções, atualizações, remoções e até mesmo os relacionamentos da base relacional no nível dos objetos, que são nossos models.

Vamos começar primeiro pelas queries e ir crescendo nosso conhecimento no decorrer deste capítulo. Para isto vamos usar nosso controller `PostsController` que se encontra dentro da pasta `Admin` em `controllers`.

Mas antes...

Os Models!

No Laravel os models são a representação, do ponto de vista de objetos, das tabelas do nosso banco de dados. Representação essa, pensando em uma entidade que represente todos os dados da tabela em questão.

Por exemplo, por convenção do framework, se eu tenho uma tabela chamada `posts` na base, a representação em model desta tabela será uma classe chamada de `Post`. Se eu tenho uma tabela `users` sua representação via model será uma classe chamada de `User`.

Quando nós temos entidades/models no singular o Laravel automaticamente tentará, por convenção, resolver sua tabela no plural por ter o pensamento, na base, de uma coleção de dados.

Por exemplo, como posso pegar todos as postagens via Model? Para isto temos um método chamada de `all` que faz este trabalho para nós.

Por exemplo:

```
1 return \App\Post::all();
```

O resultado do método acima, se fossemos pensar em uma query na base seria uma sql como esta:

```
1 select * from posts posts
```

Onde o Laravel pegará automaticamente o nome do seu model e tentará resolver ele no plural na execução da query, por exemplo model Post tabela posts.

Agora vamos conhecer o conteúdo do model Post que geramos no final do capítulo passado.

Veja abaixo:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Post extends Model
8 {
9     //
10 }
```

Veja nosso model acima, somente com sua definição, bem seca inclusive, já podemos realizar diversos trabalhos e operações em cima de nossa tabela posts associada ao model Post.

Se, por ventura, você quiser utilizar um nome de tabela de sua escolha e não quiser que o Laravel resolva o nome dela, você pode sobrescrever o atributo dentro do seu model como mostrado no conteúdo abaixo:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Post extends Model
8 {
9     protected $table = 'nome_da_sua_tabela';
10 }
```

Acima ao invés do Laravel tentar encontrar a tabela no plural ele pegará o valor do atributo \$table.

Deixarei o conteúdo acima para exemplo mas utilizarei a convenção em nossos models sem a utilização do atributo \$table. Os poderes extras dos models são concedidos pela classe Model do Eloquent.

Certo mas então o que é o Eloquent?

Eloquent?

O Eloquent é o ORM padrão do Laravel, é a camada via objetos para manipulação dos dados de seu banco. O ORM é a camada que traduz sua estrutura de objetos, Models, para a camada relacional da sua base.

Por exemplo, veja um exemplo de inserção de uma postagem utilizando o Eloquent:

```
1 //O código poderia está em um método do controller
2
3 $post = new Post();
4 $post->title      = 'Post salvo com eloquent e active
record';
5 $post->description = 'Descrição post';
6 $post->content     = 'Conteúdo do post';
```



```

7 $post->slug      = 'post-salvo-com-eloquent-e-active-
record';
8 $post->is_active  = true;
9 $post->user_id    = 1;
10
11 $post->save(); //Aqui a inserção do post com o conteúdo
acima é inserida na tabela.

```

Veja como é simples, inicio uma nova instância de `Post` chamo as colunas como atributos do objeto e por fim, para salvar os dados atribuídos a cada um dos atributos utilizo o método `save` do model para realizar a operação de criação da postagem.

Aqui no livro vou abordar mais conceitos do Eloquent de forma prática e pontuando os comportamentos. Me utilizarei, para salvar e atualizar os dados, do conceito de **Mass Assingment** que explicarei mais a frente.

PS.: Antes de prosseguirmos, recomendo fortemente a leitura sobre **Active Record** caso queira conhecer esse padrão, o modelo de inserção apresentado acima se utiliza deste padrão.

Eloquent na prática

Lembra que já temos um controller para utilização e criação de um CRUD para posts?

Por meio deste CRUD vamos focar nos pontos mais cruciais para você conhecer o trabalho do Eloquent em nossas aplicações. Nosso controller encontra-se na pasta dos controllers, dentro da pasta `Admin` e o controller `PostsController`.

Primeiramente vamos definir nosso método `index`. Trabalhando do ponto de vista dos models eu consigo realizar operações de busca dos dados em minhas tabelas, como mostrado anteriormente posso buscar todos os registros do banco de dados por meio do método `all`.

Trecho abaixo:

```
1 return \App\Post::all();
```

Ou posso retornar os dados paginados, para exibição em uma tabela html na view. Então vamos começar a implementar o método `index` no controller `PostController` que está na pasta `Admin`.

Veja o conteúdo dele abaixo:

```

1 public function index()
2 {
3     $posts = Post::paginate(15);
4
5     dd($posts); //no próximo capítulo vamos mandar para view...
6 }

```

Veja o código acima, por enquanto ainda não vamos utilizar a view, retornaremos a ela e os pontos do Blade no próximo capítulo. Voltando ao código acima, perceba que chamei o `Post::paginate` informando que quero `15` postagens, neste caso, os dados vão vir do banco paginados e teremos as `15` postagens por cada tela da paginação.

Não esqueça de importar a classe `post` em seu controller:

```
1 use App\Post;
```

Agora, vamos lá no arquivo de rotas e realizar uma pequena alteração no que já havíamos feito, para o conjunto de rotas de `post`, então, o que está assim:

```

1
Route::prefix('admin')->namespace('Admin')->group(function(){
2
3

```



```

Route::prefix('posts')->name('posts.')->group(function(){
4
Route::get('/create', 'PostController@create')->name('create')
});
5
Route::post('/store', 'PostController@store')->name('store');
6     });
7
8 });

```

Vamos atualizar para a chamada do controller como recurso, ficando como vemos abaixo:

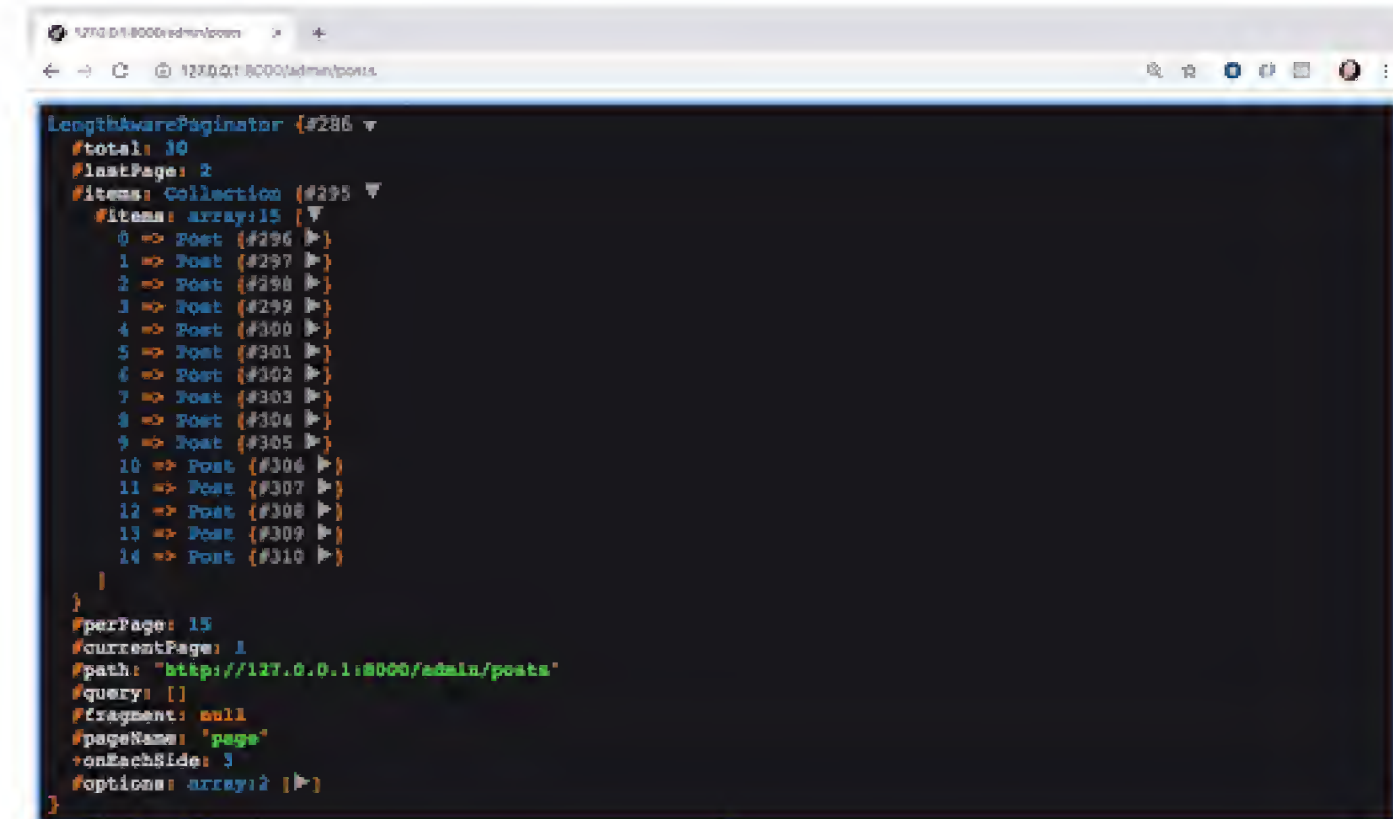
```

1
Route::prefix('admin')->namespace('Admin')->group(function(){
2
3     Route::resource('posts', 'PostController');
4
5 });

```

Perceba que agora simplificamos mais ainda nossas rotas dentro do grupo para o admin. Quando trabalhamos com request e os métodos create e store que já existem no controller tomei o cuidado de já deixá-los dentro do pensamento para o roteamento com o método resource e controllers como recurso.

Se você for ao seu browser agora e acessar `http://127.0.0.1:8000/admin/posts` você verá o resultado abaixo:



Veja que expandi o atributo items e seu nível a dentro, onde temos a coleção de dados retornada. Veja que tivemos 15 itens retornados, em nosso caso 15 postagens. Para navegar entre as páginas é bem simples, basta nós informarmos na url o parâmetro page=2 (como querystring) por exemplo.

Quando formos trabalhar com as views e o blade veremos uma forma simples de criar a paginação no frontend, simplificando esta navegação.

Buscando apenas um post

Para buscarmos dados podemos trabalhar com diversos métodos. Por exemplo, se você quiser buscar uma postagem pelo id dela, você pode usar o método find. Veja:

```

1 Post::find(1);

```


Ou ainda, buscando pelo id, você pode usar o método `findOrFail` que caso não encontre o dado em questão irá lançar uma Exception. Podemos usar desta maneira:

```
1 Post::findOrFail(1);
```

Vamos criar agora o método em nosso controller `PostController` para recuperação de uma postagem, para nossa tela de edição. Veja o conteúdo do método `show` e já adicione ele em seu controler:

```
1 public function show($id)
2 {
3     $post = Post::findOrFail($id);
4
5     dd($post); //em breve mandaremos pra view
6 }
```

Usarei o `findOrFail`, para mais a frente tratarmos melhor estas exceptions com blocos `try` e `catch`. Se você acessar a postagem de id 1 em seu browser pelo link `http://127.0.0.1:8000/admin/posts/1` você terá o resultado com a postagem de id 1 retornada.

Abaixo eu destaco só o atributo `original` que traz o dados retornados do método `findOrFail`, a postagem em questão:

```
#original: array:9 | ✓
"id" => 1
"user_id" => 7
"title" => "quia perspiciatis tenetur quis"
"description" => "Voluptatem modi iusto et officiis aperiam."
"content" => ---
    Inventore expedita incididunt consequatur ex omnia sequi. Quia maiores numquam provident voluptatem qui voluptas unde
    it ipsum ut quas impedit.\n
    \n
    Assumenda esse ut quis vero necessitatibus. Voluptates deleniti labore natus accusantium non. Ullam distinctio ut
    obis ea qui. Est et sit et n p
    \n
    Sit cupiditate ab dolor corporis. Nobis impedit mollitia et omnis consequatur. Et et sed cum distinctio tempore n
    \n
    Velit dolorem voluptatem cum vitae harum ad in. Voluptatem aut voluptate voluptates. Reque voluptatem sed magnam p
    \n
    Quia veritatis non et non. Querat laborum impedit aut possimus non aut rerum. Labore vel ullam eos fuga.\n
    \n
    Nobis numquam numquam consequatur suscipit sit quis sint. Magnam ipsa et harum perspiciatis blanditiis sit connect
    illum et culpa cum sequi vel p
    \n
    Est adipisci decimus iusto dicta nobis magni deleniti. Est facilis quis quidem. Semo cum iure aut omnis debitis bl
    \n
    Quo ut vero tempora cum recusandae voluptatum. Ut accusantium expedita corrupti animi blanditiis minima necessitat
    si assumenda consequatur eoa p
    \n
    Consequatur rem similique et repudiandae vel. Dolorum beatae praesentium rerum non corrupti repellat atque. Sunt q
    uatur tenetur dolore aut vel p
    ---
"slug" => "quo-accusamus-fuga-placeat-esse-non-esse"
"is_active" => 0
"created_at" => "2019-09-23 20:34:10"
"updated_at" => "2019-09-23 20:34:10"
```

Nosso controller até o momento está abaixo, com os dois métodos criados no capítulo sobre requests e mais o `index` e o `show`.

Veja na íntegra:

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use Illuminate\Http\Request;
6 use App\Http\Controllers\Controller;
7 use App\Post;
8
9 class PostController extends Controller
10 {
11     public function index()
12     {
13         $posts = Post::paginate(15);
14     }
```



```

15         dd($posts); //no próximo capítulo vamos mandar para
view...
16     }
17
18     public function show($id)
19     {
20         $post = Post::findOrFail($id);
21
22         dd($post);
23     }
24
25     public function create()
26     {
27         return view('posts.create');
28     }
29
30     public function store(Request $request)
31     {
32
33         if($request->hasAny(['title', 'content', 'slug'])) {
34             var_dump($request->except(['title']));
35         }
36
37         return back()->withInput();
38     }

```

Agora vamos ao método store pois nele vamos trabalhar a inserção de dados propriamente dita!

Inserindo dados com Eloquent

Como mencionei anteriormente, poderíamos utilizar o método usando **Active Record** para salvar os dados, por meio da referência dos atributos dinâmicos, baseado nas colunas do banco para aquela tabela

(posts) e por meio do método save criar um dado ou atualizar um caso quisessemos.

Como estamos tratando aqui de criação, vou mostrar a título de conhecimento o salvar dos dados usando Active Record dentro de nosso método, veja como ficaria. Altere o conteúdo do método store, já existente, para o conteúdo abaixo:

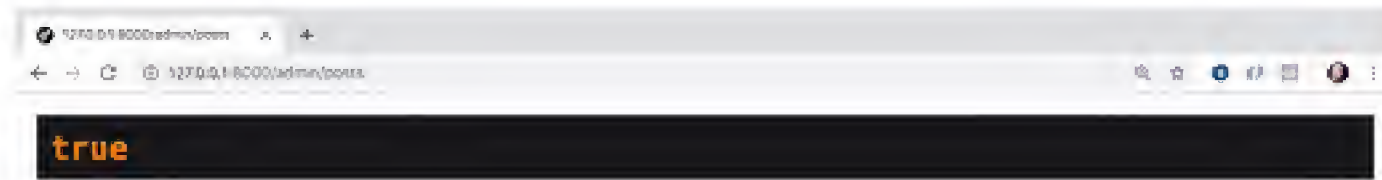
```

1 public function store(Request $request)
2 {
3     $data = $request->all();
4
5     $post = new Post();
6
7     $post->title      = $data['title'];
8     $post->description = $data['description'];
9     $post->content     = $data['content'];
10    $post->slug        = $data['slug'];
11    $post->is_active    = true;
12    $post->user_id      = 1;
13
14    dd($post->save()); //veja o resultado no browser
15 }

```

Veja que agora pego os dados de campo da request, vindas do formulário e repasso para cada atributo e na chamada do método save nós criamos este registro na base.

Para testarmos vamos ao nosso formulário no link <http://127.0.0.1/admin/posts/create> e enviar uma informação de lá. Veja o resultado na imagem abaixo:



Perceba que o resultado do método `save` foi o valor booleano `true`, confirmando assim a criação do registro em nossa base. Se você quiser atualizar um registro usando Active Record, basta, ao invés de instanciar um model `post`, passar o resultado de um `find` por exemplo:

```
1 $post = Post::find(1);
2
3 $post->title      = $data['title'];
4 $post->description = $data['description'];
5 $post->content     = $data['content'];
6 $post->slug       = $data['slug'];
7 $post->is_active  = true;
8 $post->user_id    = 1;
9
10 dd($post->save());
```

Como temos a referência na variável `$post`, agora de um dado vindo da base, ao chamarmos o método `save` o Eloquent irá atualizar este registro ao invés de criar um novo.

Este trecho foi um rápido demonstrativo do active record no Eloquent, quero te mostrar uma técnica mais direta e que é mais utilizada hoje em dia dentro do Laravel, via Eloquent. Esta técnica é o que chamamos de Mass Assignment ou Atribuição em Massa.

Vamos conhecer esta técnica.

Mass Assignment

Mass Assignment ou Atribuição em Massa é uma forma de inserir ou atualizar os dados por meio de uma única chamada e de uma vez só, como o nome já nos da lembranças.

Por exemplo eu poderia passar todo o array vindo da request e já salvar isso direto no banco por meio de um método do Eloquent, o método `create`.

Então vamos a alteração, mais uma vez do nosso método `store`, que está assim:

```
1 public function store(Request $request)
2 {
3     $data = $request->all();
4
5     $post = new Post();
6
7     $post->title      = $data['title'];
8     $post->description = $data['description'];
9     $post->content     = $data['content'];
10    $post->slug       = $data['slug'];
11    $post->is_active  = true;
12    $post->user_id    = 1;
13
14    dd($post->save()); //veja o resultado no browser
15 }
```

Agora, passará a ficar assim:

```
1 public function store(Request $request)
2 {
3     $data = $request->all();
```



```

4  $data['user_id'] = 1;
5  $data['is_active'] = true;
6
7  dd(Post::create($data));
8 }

```

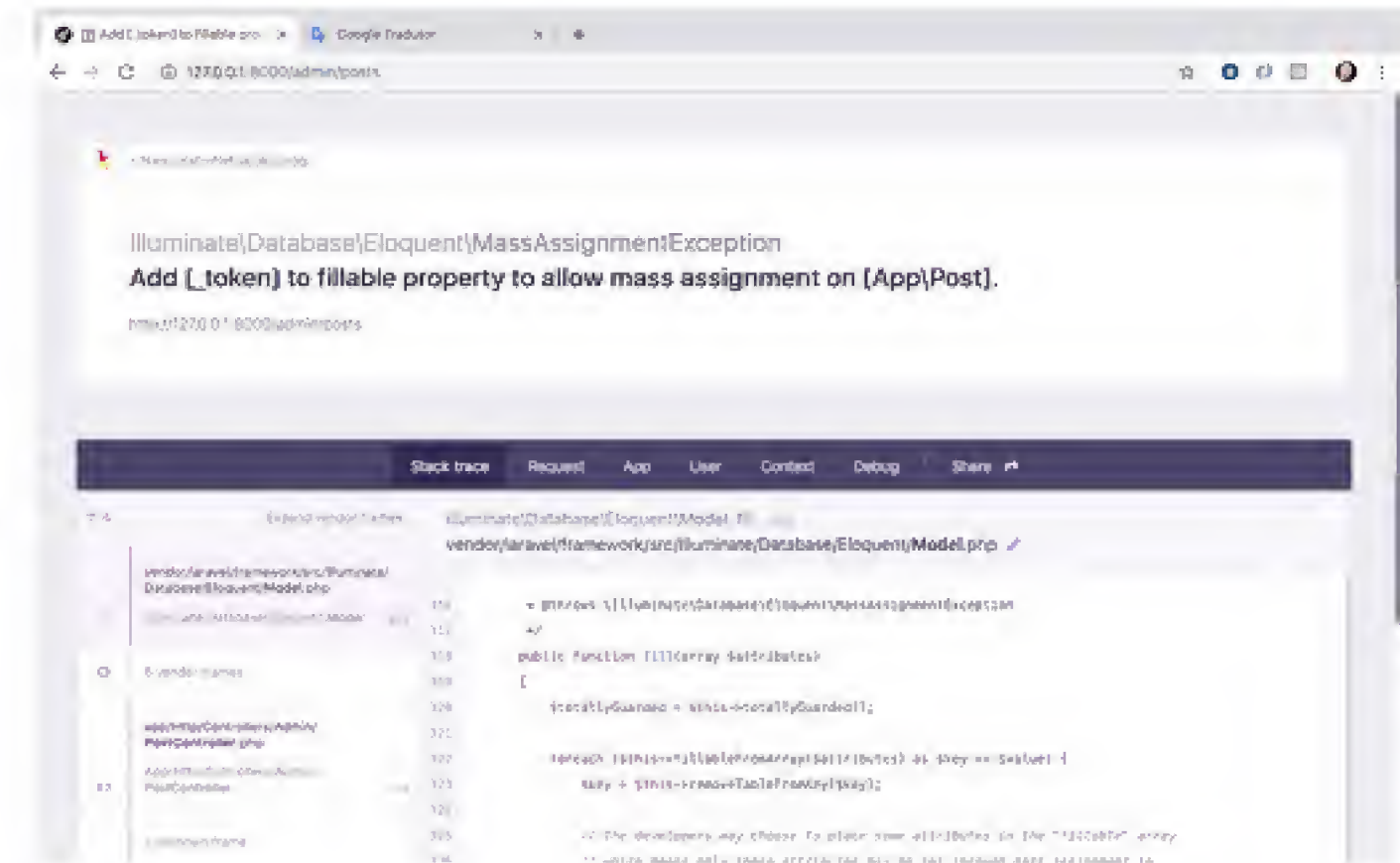
Perceba a redução acima, ao invés de chamarmos os atributos chamamos apenas o método `create` passando para ele nosso array recuperado da request. Atente só a um detalhe, o array passado pro método `create` deve respeitar, em suas chaves, os nomes das colunas da tabela em questão.

Obs.: Perceba que adicionei na mão a chave e valor para o `user_id` e a do `is_active`. Vamos trabalhar o `user_id` diretamente na parte de relação entre o Autor e a Postagem, como já mapeamos no banco. O `is_active` pode ir pro formulário com um `select` com as opções `ativo` ou `inativo` esta alteração faremos quando formos para o blade no próximo capítulo.

Se você for ao browser e testar isso enviando os dados do formulário, perceberá que teremos uma `exception` sobre a adição de um campo na propriedade `$fillable` do model, aqui entra um ponto importante.

Antes de comentar o erro, você pode está se perguntando: Esta atribuição em massa não pode ser problemática, já que ela pelo visto aceita tudo?!

Veja a `exception` lançada:



Para resolver a `exception` lançada acima, sobre o atributo `$fillable` e o seu questionamento ao mesmo tempo, nós precisamos de fato definir este bendito atributo `$fillable` lá no model, o model do momento `Post`.

Agora para que serve este atributo, tecnicamente ele é bem simples. Como estamos passando esta atribuição em massa, precisamos indicar para o Model/Eloquent que ao salvarmos os dados ou atualizarmos usando a atribuição em massa, que ele preencha somente os valores para os campos definidos no array desta propriedade, ou seja, ele só vai permitir valores para as colunas que estiverem registradas no atributo `$fillable`.

Então vamos adicionar ele em nosso model `Post` e logo após comentarmos mais um pouco sobre este detalhe.

Veja a alteração em `Post.php`:

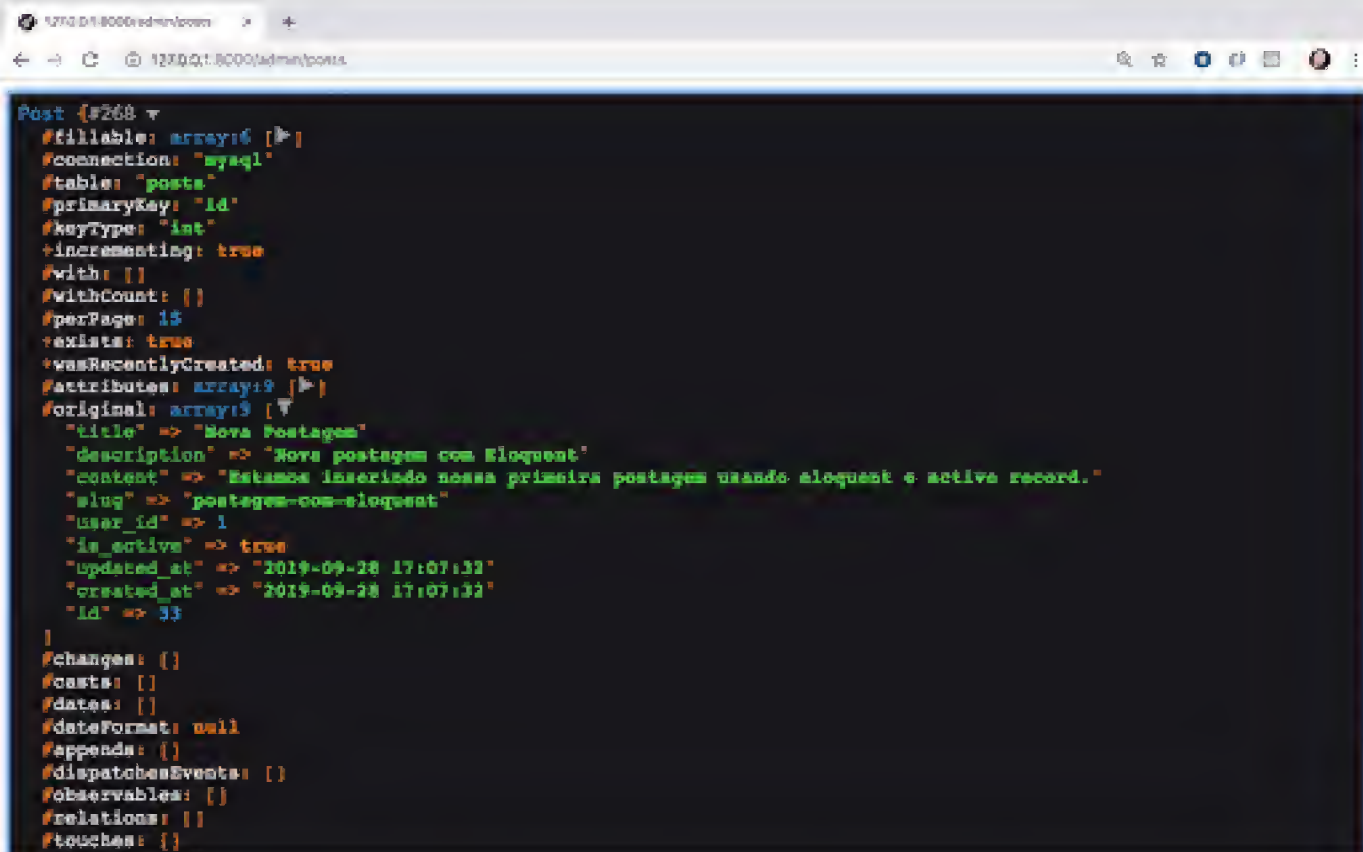

```

1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Post extends Model
8 {
9     protected $fillable = [
10         'title',
11         'description',
12         'content',
13         'slug',
14         'is_active',
15         'user_id'
16     ];
17 }

```

Agora com os campos adicionados no atributo \$fillable vamos enviar os dados novamente do nosso formulário.

Veja o resultado, no dd, vindo do método create:



```

Post [#268 ▼]
#fillable: array:6 [▶]
#connection: "mysql"
#table: "posts"
#primaryKey: "id"
#keyType: "int"
+incrementing: true
#with: []
#withCount: []
#perPage: 15
+exists: true
+wasRecentlyCreated: true
#attributes: array:9 [▶]
#original: array:9 [▼]
  "title" => "Nova Postagem"
  "description" => "Nova postagem com Eloquent"
  "content" => "Estamos inserindo nossa primeira postagem usando eloquent e active record."
  "slug" => "postagem-com-eloquent"
  "user_id" => 1
  "is_active" => true
  "updated_at" => "2019-09-28 17:07:32"
  "created_at" => "2019-09-28 17:07:32"
  "id" => 33
]
#changes: {}
#casts: []
#dates: []
#dateFormat: null
#appends: []
#dispatchesEvents: []
#observables: []
#relations: []
#touches: []

```

O método create ao criar um dado, retorna este dado criado junto com seu id na base como resultado. Veja o conteúdo da informação abrindo a propriedade original.

A segurança do método create, usando a atribuição em massa, se dá pela propriedade \$fillable no model que uma vez definida e tendo as colunas permitidas só teremos o preenchimento das informações para a coluna mapeada nesta propriedade.

Agora como fazemos a atualização do dados massa?

Vamos lá.

Atualizando Dados em Massa

Para atualizarmos os dados vamos trabalhar aqui com nossa view de edição e conhecer mais alguns detalhes do Laravel, crie lá dentro da

pasta resources/views/posts o arquivo edit.blade.php e adicione o conteúdo abaixo:

```

1 <form action="
  {{route('posts.update', ['postId' => $post->id])}}" method="post">
2
3     @csrf
4     @method("PUT")
5
6     <div class="form-group">
7         <label>Titulo</label>
8         <input type="text" name="title" class="form-
9 control" value="{{ $post->title }}"
10     ">
11
12     <div class="form-group">
13         <label>Descrição</label>
14         <input type="text" name="description" class="form-
15 control" value="{{ $post->description }}"
16     ">
17
18     <div class="form-group">
19         <label>Conteúdo</label>
20
21     <textarea name="content" id="" cols="30" rows="10" class="form-control">{{ $post->content }}</textarea>
22
23
24     <div class="form-group">
25         <label>Slug</label>

```

```

26         <input type="text" name="slug" class="form-
27 control" value="{{ $post->slug }}"
28     ">
29     <button class="btn btn-lg btn-success">Atualizar
30     Postagem</button>
31 </form>

```

Agora, lá no método show do PostController substitua a linha do dd pelo trecho abaixo:

```
1 return view('posts.edit', compact('post'));
```

Se você acessar o link <http://127.0.0.1:8000/admin/posts/1> você obterá o resultado abaixo:

Titulo quis perspiciatis teneti

Descrição Voluptatem modi iusto

Contento Inventore expedita incidunt consequatur ex omnis sequi. Quis maiores numquam provident voluptatem qui voluptas unde sint. Quam impedit ipsum ut quas impedit. Assumenda esse ut quia vero necessitatibus. Voluptates deleniti labore natus

Slug quo-accusamus-fuga-

Criar Postagem

Veja nosso formulário de edição acim já preenchido com os valores

pegos do banco pelo Eloquent e enviados para a view.

Agora vamos entender os códigos do formulário de edição acima. Vamos lá.

Primeiramente atente a chamada da rota na action do formulário:

```
1 <form action="
  {{route('posts.update', ['post' => $post->id])}}"
  method="post">
```

Enviaremos nossos dados para a rota de apelido `posts.update` atribuída pelo método `resource` do Route, informamos o nome do parâmetro dinâmico da rota no array do segundo parâmetro da função helper `route` que será o `id` da postagem, estes dados serão enviados para o método `update` lá do `PostController`, que vamos criar.

Temos agora mais uma alteração/novidade, a chamada da diretiva `@method`. Vamos entender ela:

Sabemos que os formulários html só suportam os verbos http: `post` e `get`. Por meio da diretiva `@method` fazemos o Laravel interpretar o formulário em questão com o verbo definido na diretiva, ou seja, como usei o valor `PUT` este formulário será interpretado pelo Laravel como sendo enviado via verbo **PUT** e cairá para a execução do método `update` do controller, que é o método que recebe as solicitações quando usamos os verbos `PUT` ou `PATCH` em nossa requisição.

Continuando, agora em nossos inputs recebemos os valores vindos lá do controller. Quando fazemos uma busca pela postagem desejada, ele retorna um objeto populado com os dados desta postagem, o que nos resta é acessarmos eles respeitando os nomes das colunas mas chamando como atributos do objeto e isto pode ser visto em cada atributo `value` dos inputs do nosso formulário de edição.

Agora precisamos definir o método para manipulação do dados enviados do formulário de edição, para isso crie um método chamado `update` em seu controller com o conteúdo abaixo:

```
1 public function update($id, Request $request)
2 {
3     //Atualizando com mass assignment
4     $data = $request->all();
5
6     $post = Post::findOrFail($id);
7
8     dd($post->update($data));
9 }
```

Perceba que busquei a postagem usando o método `findOrFail` pelo `id` vindo da url, o parâmetro dinâmico. Neste caso como quero atualizar, o método da atribuição para atualização em massa, é o método do eloquent o `update` que me retorna um booleano para o sucesso ou falha desta execução.

Acessando o formulário de edição no link `http://127.0.0.1:8000/admin/posts/1` vamos alterar algum campo e clicar em Atualizar Postagem. Veja o resultado:



Vamos ao passo final do CRUD, a remoção de um dado com Eloquent.
Deletando Registros

Bom para completarmos nosso ciclo, vamos deletar postagens de nossa base. Isso será realizado pelo método `delete` do Eloquent. Veja o método abaixo, a ser adicionado no `PostController`:

```
1 public function destroy($id)
2 {
3     $post = Post::findOrFail($id);
4
5     dd($post->delete());
6 }
```

O método do controller como recurso que responde a chamada para remoção de um dado é o `destroy`, entretanto, o método do Eloquent que irá remover um dado é o `delete` que retorna um booleano para o caso de sucesso ou falha da operação de remoção do dado da base.

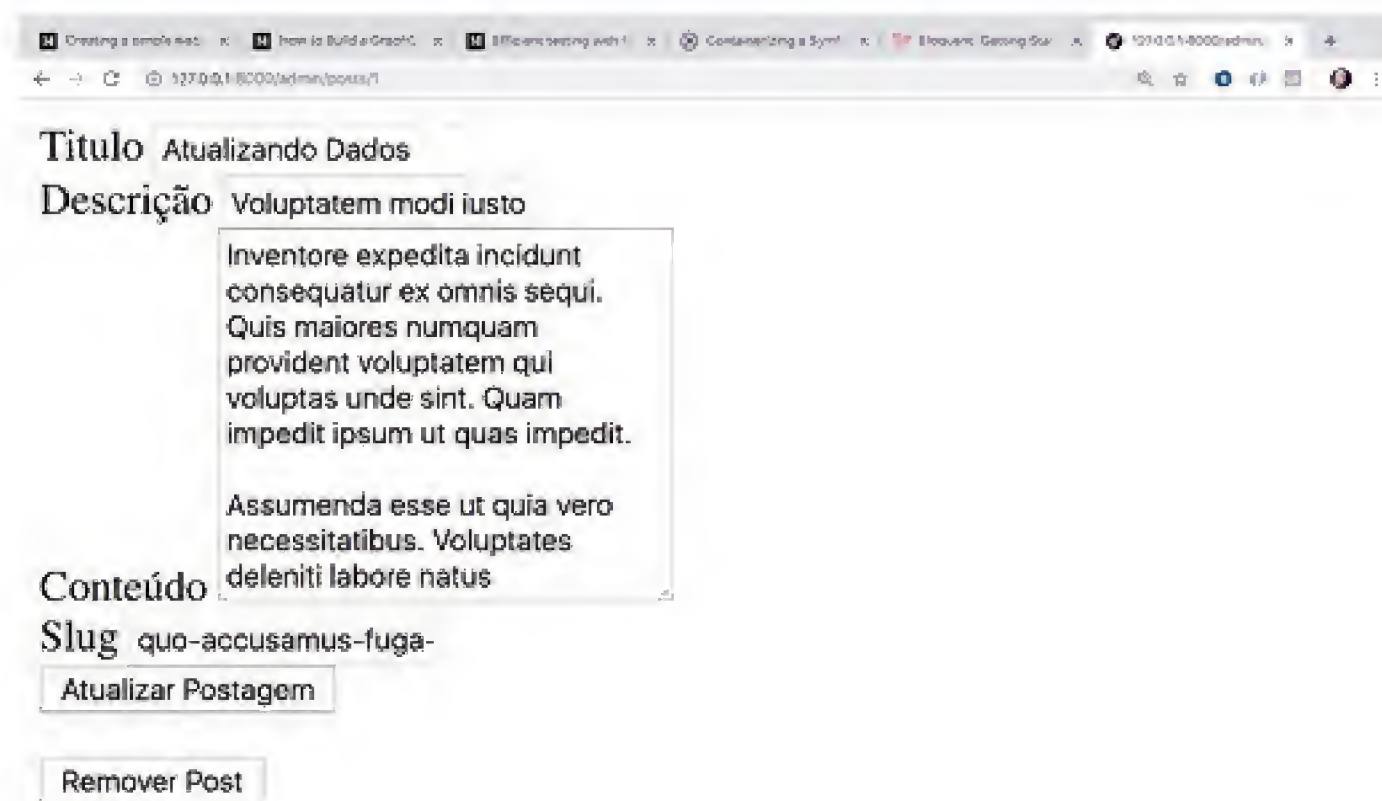
Perceba que como queremos remover uma postagem, precisamos buscar por ela via Eloquent para então removermos.

Agora precisamos adicionar o botão de remoção lá na tela de edição, neste caso como precisamos simular o envio do verbo http **DELETE**, vou precisar usar um form para o botão de remoção. Adicione o form abaixo, após o formulário de edição:

```
1 <form action="
  {{route('posts.destroy', ['post' => $post->id])}}" method="po
  st">
2     @csrf
3     @method('DELETE')
4     <button type="submit">Remover Post</button>
5 </form>
```

Perceba também que temos adicionado o controle **csrf** para esta operação além da definição da diretiva `@method` com o verbo http **DELETE**.

A tela de edição fica assim:



Ao clicar no botão remover você verá o resultado na tela, **true** para sucesso na remoção ou **false** para o caso de falha. Após removido se tentarmos acessar a mesma postagem teremos uma tela de 404 em nossa cara:



Agora completamos as 4 etapas de um CRUD completo utilizando o Eloquent que nos permite trabalhar com o banco pela visão de objetos.

Conclusões

Neste capítulo conhecemos diversas possibilidades usando o Eloquent, o ORM padrão do Laravel. Com isso realizamos a criação de um CRUD completo usando nosso Model Post por meio dos métodos disponíveis e herdados do Model do Eloquent, que nos permitiu realizarmos estas operações de seleção, criação, atualização e ainda remoção das postagens no banco de dados.

Para completarmos o ciclo e deixarmos as coisas mais dinâmicas e integradas precisamos conhecer o **Blade**, o famoso *template engine* do Laravel e que nos permite escrever views de forma mais dinâmica e rápida.

Conheceremos o Blade no próximo capítulo. Até lá!

Blade, Template Engine do Laravel

Olá tudo bem? Vamos continuar nosso livro e desta vez vamos conhecer o famoso template engine do Laravel, o Blade.

O Blade é o template engine default do Laravel e traz consigo diversas estruturas que simplificam muito nosso trabalho na criação de nossas views, estruturas condicionais, laços e também diversas diretivas que nos permitem criar controles de forma simples e rápida, por conta da pouca escrita para alcançar um resultado desejado.

O Blade ao longo do tempo veio e vem recebendo diversos incrementos e pretendo mostrar, na prática, estas possibilidades usando seu poder para construirmos nossas views.

Então vamos criar as views de nossa aplicação utilizando o poder do Blade!

Layouts

Vamos começar pelo layout, em nosso Hello World com Laravel nós tivemos um contato com o blade quando mandamos nossa mensagem (Hello World) para a view e exibimos ela com o interpolador `{{ }}`, que nos permite retornar/exibir informações passadas a ele. Antes de entrarmos em pontos como este que comentei, vamos definir e organizar um template base usando o blade para melhor dispormos as views do nosso painel.

Primeiramente crie uma pasta chamada de layouts dentro da pasta de views e dentro da pasta layouts crie o arquivo `app.blade.php`. Adicione o seguinte conteúdo e logo após farei os comentários:


```

1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no,
initial-scale=1.0, maximum-\
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Gerenciador de Posts</title>
10
11     <link rel="stylesheet" href="https://stackpath.bootstrapcdn.
com/bootstrap/4.3.1/\
12 css/bootstrap.min.css">
13 </head>
14 <body>
15     <nav class="navbar navbar-expand-lg navbar-light bg-
light">
16         <a class="navbar-brand" href="/">Laravel 6 Blog</a>
17         <button class="navbar-toggler" type="button" data-
toggle="collapse" data-tar\
18 get="#navbarNavDropdown" aria-
controls="navbarNavDropdown" aria-expanded="false" ari\
19 a-label="Toggle navigation">
20             <span class="navbar-toggler-icon"></span>
21         </button>
22         <div class="collapse navbar-
collapse" id="navbarNavDropdown">
23             <ul class="navbar-nav">
24                 <li class="nav-item active">
25                     <a class="nav-
link" href="{ route('posts.index') }">Posts</a>

```

```

25                 </li>
26             </ul>
27         </div>
28     </nav>
29     <div class="container">
30         @yield('content')
31     </div>
32 </body>
33 </html>

```

Acima temos a definição do nosso layout base, aqui entramos no primeiro conceito do Blade, neste caso, na herança de templates. Se você perceber no layout, defini dentro do body na div.container uma diretiva chamada de @yield que me permite apontar onde os templates, que irão herdar deste layout, devem exibir seus conteúdos.

Por exemplo, vamos entender como isso acontece fazendo nossa view create.blade.php herdar do nosso layout app.blade.php. Veja as alterações que fiz na view create.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{ route('posts.store') }}" method="post">
5
6         @csrf
7
8         <div class="form-group">
9             <label>Titulo</label>
10             <input type="text" name="title" class="form-
control" value="{{ old('title') }}">
11         </div>
12
13

```



```

14     <div class="form-group">
15         <label>Descrição</label>
16
17         <input type="text" name="description" class="form-
18         control" value="{{old(\
19         'description')}}">
20     </div>
21
22     <div class="form-group">
23         <label>Conteúdo</label>
24
25         <textarea name="content" id="" cols="30" rows="10" class="fo
26         rm-control">\
27         {{old('content')}}</textarea>
28     </div>
29
30     <div class="form-group">
31         <label>Slug</label>
32         <input type="text" name="slug" class="form-
33         control" value="{{old('slug')}}\
34         }}">
35     </div>
36
37     <button class="btn btn-lg btn-
38     success">Criar Postagem</button>
39 </form>
40 @endsection

```

Aqui temos o conteúdo do formulário envolvido por uma diretiva chamada de @section que recebe o valor content e a definição de onde essa diretiva termina com o @endsection. O que isso quer dizer?!

A diretiva @section define o conteúdo que será substituído no layout principal, ou seja, quando eu acessar essa view ele vai herdar o que tem

em app.blade.php e onde eu defini o @yield('content') será adicionado o conteúdo que temos na diretiva @section, no caso da view create.blade.php exibirá o conteúdo do formulário.

Mas Nanderson, onde está definido que o create.blade.php herda de layout.blade.php? Não comentei de propósito mas ele se encontra como sendo a primeira linha da nossa view, veja a definição que aponta de qual template create.blade.php herda por meio da diretiva @extends que recebe o layout pai da view em questão. Neste caso digo que a view create.blade.php herda de app.blade.php informando a diretiva @extends da seguinte maneira: @extends('layouts.app').

Sendo que layouts é a pasta dentro de views e app o arquivo app.blade.php, onde sabemos que o Laravel irá incluir a extensão internamente e linkar o caminho completo até a pasta layouts.

Altere também a view edit.blade.php, segue o conteúdo:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{route('posts.update', ['post' =>
5     $post->id])}}" method="post">
6
7         @csrf
8         @method("PUT")
9
10        <div class="form-group">
11            <label>Titulo</label>
12            <input type="text" name="title" class="form-
13            control" value="{{ $post->tit\
14            le}}">
15        </div>
16
17        <div class="form-group">

```



```

16         <label>Descrição</label>
17
18         <input type="text" name="description" class="form-
19         control" value="{{ $post->description }}">
20
21         <div class="form-group">
22             <label>Conteúdo</label>
23
24             <textarea name="content" id="" cols="30" rows="10" class="fo
25             rm-control">{{ $post->content }}</textarea>
26
27             <div class="form-group">
28                 <label>Slug</label>
29                 <input type="text" name="slug" class="form-
30                 control" value="{{ $post->slug }}">
31
32             </div>
33
34             <button class="btn btn-lg btn-
35             success">Atualizar Postagem</button>
36
37         </form>
38         <hr>
39         <form action="{{ route('posts.destroy', ['post' =>
40         $post->id]) }}" method="post">
41             @csrf
42             @method('DELETE')
43             <button type="submit" class="btn btn-lg btn-
44             danger">Remover Post</button>
45         </form>

```

```
41 @endsection
```

Obs.: Fiz duas pequenas alterações além da definição da section e do extends. Adicionei a tag html hr entre o form de edição e o do botão de Remover Post e adicionei no button de remover posts as classes: btn btn-lg btn-danger.

Veja o formulário na íntegra na imagem abaixo:

The screenshot shows a web form for managing blog posts. At the top, there's a header 'Laravel 6 Blog' and a 'Posts' link. The form has four main sections: 'Título' (Title) with a text input, 'Descrição' (Description) with a text input, 'Conteúdo' (Content) with a large text area, and 'Slug' with a text input. Each section has a label above it. Below the form, there are two buttons: a green 'Atualizar Postagem' button and a red 'Remover Post' button. The form is styled with Bootstrap classes, including 'form-group' for each section and 'btn btn-lg btn-success' for the update button.

Lembra que comentei que já tinha adicionado algumas classes do Bootstrap no Formulário, agora que linkamos o bootstrap.css lá no app.blade.php os estilos foram aplicados e nossa interface está mais aceitável.

Agora vamos para a nossa listagem dos posts e conhecer mais possibilidades do Blade.

Laços de Repetição & Condicionais

Sabemos que para iterarmos em cima de uma coleção de dados

precisamos usar laços de repetição, o Blade nos traz algumas possibilidades interessantes. A primeira delas é a possibilidade de utilização do foreach, como vemos abaixo:

```
1 @foreach($posts as $post)
2     <li>{{ $post->title }}</li>
3 @endforeach
```

Desta maneira a iteração na coleção de posts vindas do banco de dados, será no mesmos moldes do foreach do PHP, e claro, no processamento desta view essa diretiva se tornará um foreach nativo.

Mas aqui quero utilizar um foreach não muito convencional do dia a dia da forma que vamos escrever, mas é claro que com condicionais e combinando com os laços chegaríamos no mesmo resultado. Mas Nanderson, do que você está falando!

Por exemplo, poderíamos fazer um controle condicional pro caso de não existirem postagens na base e somente, se existirem, exibissemos a tabela com as postagens.

Por exemplo, veja o trecho abaixo:

```
1 @if($posts)
2     @foreach($posts as $post)
3         <li>{{ $post->title }}</li>
4     @endforeach
5 @else
6     <h2> Nenhuma postagem cadastrada!</h2>
7 @endif
```

Acima de cara te apresento o controle condicional ou como usar os senão (if...else) via diretivas do Blade. Primeiramente verificamos se o valor de \$posts é verdadeiro, se verdadeiro nós realizamos os loops,

senão, exibimos uma mensagem padrão.

Agora podemos melhorar ainda mais essa escrita usando blade, com a diretiva de loop chamada de @forelse. É ela que vamos utilizar então vamos ao conteúdo do nosso index.blade.php, que não existe ainda por isso crie o arquivo index.blade.php lá dentro da pasta das views das postagens.

Veja seu conteúdo abaixo:

```
1 @extends('layouts.app')
2
3 @section('content')
4     <div class="row">
5         <div class="col-sm-12">
6             <a href="{{ route('posts.create') }}" class="btn btn-success float-right">\
7 Criar Postagem</a>
8             <h2>Postagens Blog</h2>
9             <div class="clearfix"></div>
10        </div>
11    </div>
12    <table class="table table-striped">
13        <thead>
14            <tr>
15                <th>#</th>
16                <th>Titulo</th>
17                <th>Status</th>
18                <th>Criado em</th>
19                <th>Ações</th>
20            </tr>
21        </thead>
22        <tbody>
23            @forelse($posts as $post)
```



```

24         <tr>
25             <td>{{ $post->id }}</td>
26             <td>{{ $post->title }}</td>
27             <td>
28                 @if($post->is_active)
29                     <span class="badge badge-
success">Ativo</span>
30                 @else
31                     <span class="badge badge-
danger">Inativo</span>
32                 @endif
33             </td>
34             <td>{{ date('d/m/Y H:i:s', strtotime($post->created_at)) }}</td>
35             <td>
36                 <div class="btn-group">
37                     <a href="{{ route('posts.show',
['post' => $post->id]) }}" class="btn
38 ss="btn btn-sm btn-primary">
39                         Editar
40                     </a>
41                     <form action="{{
{{ route('posts.destroy', ['post' => $post->id]) }}
42 }}" method="post">
43                         @csrf
44                         @method('DELETE')
45                     <button type="submit" class="btn btn-sm btn-danger">Remo\
46 ver</button>
47                     </form>
48                 </div>

```

```

49             </td>
50         </tr>
51         @empty
52             <tr>
53                 <td colspan="4">Nada encontrado!</td>
54             </tr>
55         @endforelse
56     </tbody>
57 </table>
58 @endsection

```

Não esqueça de substituir, lá no controller PostController, no método index, a linha do dd pela linha abaixo:

```
1 return view('posts.index', compact('posts'));
```

Vamos aos pontos da view index.blade.php. Perceba que aqui usei a diretiva:

```

1 @forelse($posts as $post)
2     ...
3 @empty
4     ...
5 @endforelse

```

Para iterar as postagens vindas do controller, neste caso o @forelse fará o seguinte: Se não houverem postagens na base ele cairá na execução do bloco @empty, onde adicionamos um tr com td e a mensagem Nada Encontrado!. Veja o bloco do @empty abaixo:

```

1 @empty
2     <tr>
3         <td colspan="4">Nada encontrado!</td>
4     </tr>

```


5 @endforelse

Existindo posts, que é o nosso caso, teremos a exibição da tabela com as postagens. Vamos analisar o bloco do @forelse.

Veja somente ele abaixo:

```

1 @forelse($posts as $post)
2     <tr>
3         <td>{{ $post->id }}</td>
4         <td>{{ $post->title }}</td>
5         <td>
6             @if($post->is_active)
7                 <span class="badge badge-
success">Ativo</span>
8             @else
9                 <span class="badge badge-
danger">Inativo</span>
10            @endif
11        </td>
12        <td>{{ date('d/m/Y H:i:s', strtotime($post->created_at)) }}</td>
13        <td>
14            <div class="btn-group">
15                <a href="{{ route('posts.show',
['post' => $post->id]) }}" cla\
16 ss="btn btn-sm btn-primary">
17                    Editar
18                </a>
19                <form action="
{{ route('posts.destroy', ['post' => $post->id])\
20 }}" method="post">

```

```

21                @csrf
22                @method('DELETE')
23            <button type="submit" class="btn btn-sm btn-danger">Remo\
24 ver</button>
25        </form>
26    </div>
27 </td>
28 </tr>
29 ...

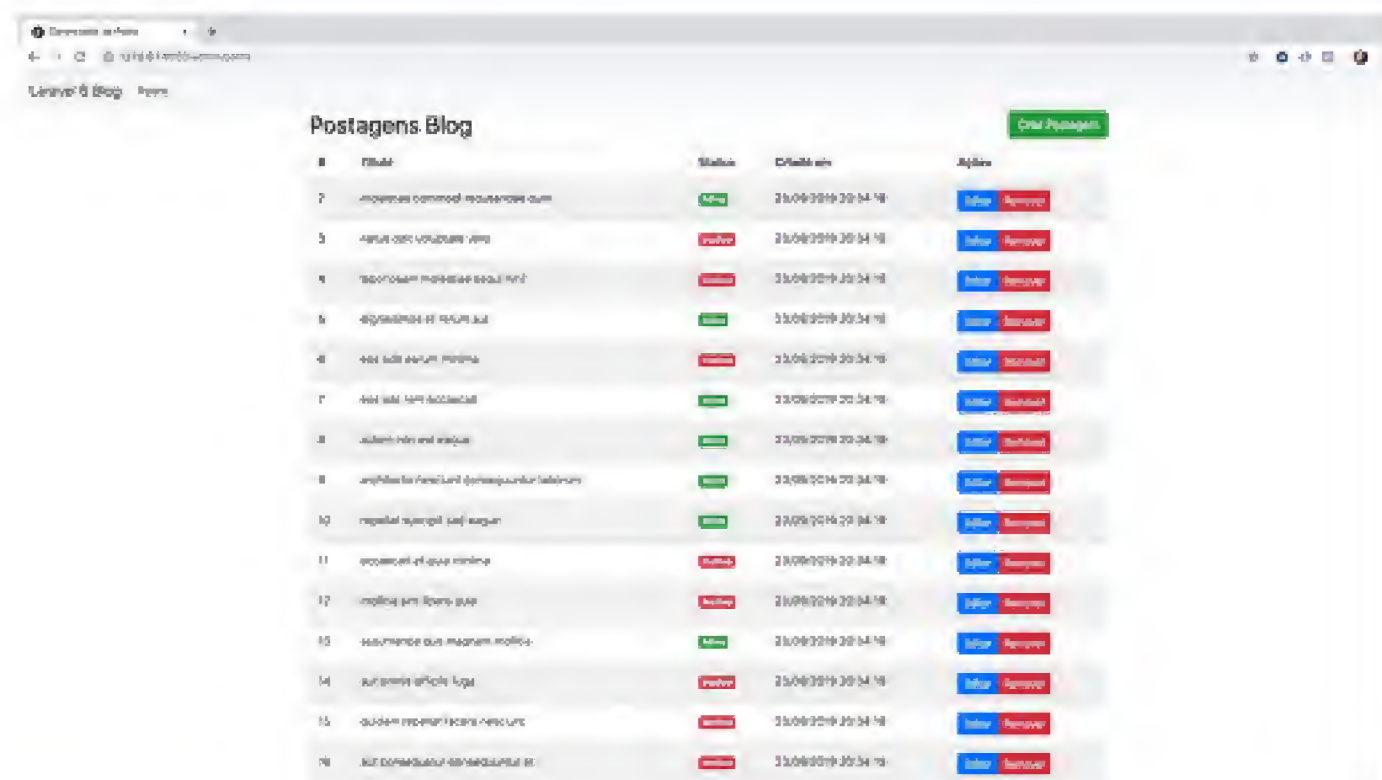
```

O que vale destacar aqui é a utilização da diretiva de controle condicional para exibição da mensagem Ativo ou Inativo dentro de um badge provido pelo Bootstrap.css, usei também, no campo da data de criação da postagem, a função date do PHP para formatação.

Temos ainda, para a coluna de **Ações**, o botão de edição onde linkamos a rota de edição usando o apelido dela por meio da função helper route e passando o id dinâmico do post como segundo parâmetro.

Sobre o botão da remoção do post, apenas peguei o form que tínhamos criado para remoção da postagem no último capítulo alterando apenas a classe que estava btn-lg(para botões grandes) e coloquei btn-sm(para botões pequenos), além de agrupar os botões de edição e da remoção usando a div com a classe do bootstrap btn-group.

Veja o resultado da nossa tela de listagem das postagens:



Veja que temos apenas 15 postagens nesta tela, agora como fazer para exibirmos a paginação abaixo da tabela para navegação entre as telas da paginação?!

Vamos lá!

Paginação na View

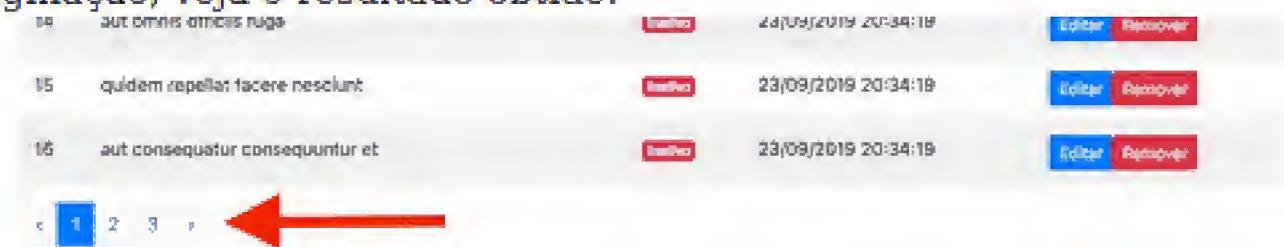
Como usamos o método `paginate` lá no nosso controller, podemos por meio da coleção de posts que recebemos na view, dentro da variável `$post`, exibir a paginação de forma bem simples e direta!

Adicione o trecho abaixo, logo após o fechamento da tag `table` da sua view `index.blade.php`:

```
1 {{ $posts->links() }}
```

O seguinte trecho exibirá nosso html de navegação entre cada tela da

paginação, veja o resultado obtido:



Simples assim, agora já temos uma paginação tanto nos dados vindos do banco quando na exibição dos links da navegação de forma simples e direta!

Conclusões

Neste capítulo nós fechamos o ciclo MVC iniciado lá no começo do livro. Além de termos tido contato com diversas diretivas do blade, que juntadas as que vimos aqui já te dão uma excelente base de aprendizado e aplicação!

É claro que ainda veremos mais opções do blade mas isso vou diluindo no decorrer dos próximos capítulos. Agora que temos nosso CRUD completo, em se tratando de persistência no banco e retornando para as views os resultados, além de termos dado um tapa nas nossas views vamos para o próximo capítulo onde vamos trabalhar os relacionamentos da base do ponto de vista dos models.

Vamos lá!

Relacionamentos com Eloquent

Olá, tudo bem? Espero que sim!

Vamos começar agora um trecho que é necessário bastante atenção e cuidado, vamos falar sobre relacionamentos de uma base relacional (nosso banco de dados) e a representação destes relacionamentos do ponto de vista de Objetos, representados por nossos Models junto com o Eloquent.

Como venho fazendo vamos mostrar os relacionamentos e suas nuances aplicados em nosso blog onde teremos as seguintes representações:

- Relacionamento 1:N(Um para Muitos): Autor x Postagens;
- Relacionamento N:N(Muitos para Muitos: Postagens & Categorias;

Já temos definido na base o primeiro relacionamento da listagem acima e vamos começar por ela. Para o relacionamento entre postagem e categoria, vou criar os insumos(Models, Controllers e etc) no momento que forem necessários, até para darmos uma lembrada.

Então vamos lá, mãos a obra! Quer dizer mão ao teclado!

Relacionamento 1:N (Um para Muitos e Inverso)

Primeiramente vamos definir nossa relação entre Autor(**User**) e suas Postagens(**Post**). Para isto precisamos definir métodos dentro de cada model que representem esta ligação.

Primeiramente vamos criar o método do ponto de vista de Post em relação ao nosso User. Veja o método abaixo adicionado ao model Post:

```
1 public function user()
2 {
```

```
3     return $this->belongsTo(User::class);
4 }
```

Definimos o método acima para representar a ligação entre nossos models, neste caso, quando precisarmos acessar dados desta relação ou criar um dados vamos chamar o método user do ponto de vista de Post.

O método belongsTo vêm do Eloquent e indica que o model Post pertence a(**belongsTo**) User, por isso passei o nome qualificado do model User no primeiro parâmetro do método belongsTo.

Outro ponto a ressaltar aqui é que, o Laravel vai tentar resolver o nome da coluna referência na tabela **posts** por meio do nome do método, se coloquei user ele vai buscar dentro da tabela post, no banco de dados, pela referência user_id.

Se por ventura você usou um nome de coluna, que representa a referência na sua tabela diferente, por exemplo, não usou o user_id mas sim author_id. Neste caso você precisa informar para o Eloquent o nome da coluna, veja um trecho exemplo abaixo:

```
1 return $this->belongsTo(User::class, 'author_id');
```

Desta forma quando o Eloquent for acessar suas tabelas e gerar as queries sql dos acessos irá buscar pela coluna author_id. Lembrando, isso vale apenas para o nome da coluna, que recebe a chave estrangeira, que não seja user_id em nosso caso.

Agora como digo que o model User têm muitas postagens? Vamos lá no model User e vamos definir o método abaixo:

```
1 public function posts()
2 {
3     return $this->hasMany(Post::class);
4 }
```


Se indiquei que o Post pertence a User por meio do método **belongsToMany**, dentro de User eu indico que ele tem muitos(hasMany) posts por meio do método hasMany, informando também o nome do model no primeiro parâmetro, neste caso Post. Agora toda vez que eu precisar acessar as postagens de um usuário, eu irei acessar o método posts para tal processo.

As definições do ponto de vista do Model são estas, agora, para entendermos melhor vamos realizar algumas queries para testarmos esta relação.

Vamos exibir lá na listagem de posts o autor da postagem. Para isto adicione mais uma coluna no thead, depois da coluna do id(#):

```
1 <th>Autor</th>
```

No tbody vamos adicionar o conteúdo para esta coluna, veja abaixo:

```
1 <td>{{ $post->user->name }}</td>
```

Antes de vermos o resultado no browser, vamos ver na íntegra a view de posts agora, com esta alteração:

```
1 @extends('layouts.app')
2
3 @section('content')
4     <div class="row">
5         <div class="col-sm-12">
6             <a href="{{ route('posts.create') }}" class="btn
7 btn-success float-right">\
8             Criar Postagem</a>
9             <h2>Postagens Blog</h2>
10             <div class="clearfix"></div>
11         </div>
12     </div>
```

```
12 <table class="table table-striped">
13     <thead>
14         <tr>
15             <th>#</th>
16             <th>Autor</th>
17             <th>Titulo</th>
18             <th>Status</th>
19             <th>Criado em</th>
20             <th>Ações</th>
21         </tr>
22     </thead>
23     <tbody>
24         @foreach($posts as $post)
25             <tr>
26                 <td>{{ $post->id }}</td>
27                 <td>{{ $post->user->name }}</td>
28                 <td>{{ $post->title }}</td>
29                 <td>
30                     @if($post->is_active)
31                         <span class="badge badge-
32 success">Ativo</span>
33                     @else
34                         <span class="badge badge-
35 danger">Inativo</span>
36                     @endif
37                 </td>
38                 <td>
39                     <div class="btn-group">
40                         <a href="{{ route('posts.show',
```



```

[ 'post' => $post->id] ] } }" class="btn btn-sm btn-primary">
40 ss="btn btn-sm btn-primary">
41         Editar
42     </a>
43     <form action="
44 {{route('posts.destroy', ['post' => $post->id])\
45 }}" method="post">
46         @csrf
47         @method('DELETE')
48     <button type="submit" class="btn btn-sm btn-danger">Remo\
49 ver</button>
50 </form>
51 </div>
52 </td>
53 </tr>
54 @empty
55 <tr>
56     <td colspan="4">Nada encontrado!</td>
57 </tr>
58 @endforelse
59 </tbody>
60 </table>
61 {{ $posts->links() }}
62 @endsection

```

Veja o resultado:

Laravel 6 Blog Posts

Postagens Blog

Criar Postagem

#	Autor	Título	Status	Criado em	Ações
2	Dr. Karley Mitchell	molestias commodi recusandae cum	Ativo	23/09/2019 20:34:19	Editar Remover
3	Jan Rippin	natus odit voluptate vero	Inativo	23/09/2019 20:34:19	Editar Remover
4	Genevra Franecki V	laboriosam molestiae sequi nihil	Inativo	23/09/2019 20:34:19	Editar Remover
5	Dr. Karley Mitchell	dignissimos et rerum aut	Ativo	23/09/2019 20:34:19	Editar Remover
6	Dr. Dayana Price	eos odit earum minima	Inativo	23/09/2019 20:34:19	Editar Remover
7	Destini Cormier DDS	eos iste rem occaecati	Ativo	23/09/2019 20:34:19	Editar Remover
8	Reynold Schaefer	autem non est eaque	Ativo	23/09/2019 20:34:19	Editar Remover
9	Reynold Schaefer	architecto nesciunt consequuntur laborum	Ativo	23/09/2019 20:34:19	Editar Remover
10	Genevra Franecki V	repellat suscipit sed eaque	Ativo	23/09/2019 20:34:19	Editar Remover
11	Dr. Dayana Price	occaecati et quia minima	Inativo	23/09/2019 20:34:19	Editar Remover

Por meio da definição do método `user` em `Post` e `posts` em `User` criamos as relações entre os objetos dos dois pontos de vistas. Como precisei exibir o autor da postagem precisei chamar o método `user` e assim consigo acessar os atributos deste `user`, associado ao `post` em questão, onde por exemplo peguei o nome dele:

```
1 {{ $post->user->name }}
```

Agora você pode está se perguntando, cara tu acessou `user` como se fosse um atributo mas tu definiu um método lá no `model`?!!

Vou explicar isso agora, vamos lá!

Aqui tecnicamente é bem simples, quando você acessar a ligação como se fosse o atributo da classe uma coleção é retornada, especialmente quando usamos o `hasMany` e o `belongsToMany` (veremos este método na próxima sessão) ou o objeto representado o `model` será retornado, pro caso especial do método `belongsTo` ou `hasOne` (veremos este método no

próximo capítulo) da ligação.

Para simplificar:

- Como user tem muitos posts o retorno de `$user->posts` seria a coleção de posts ligadas ao usuário em questão;
- Já como uma postagem está ligada ou pertence a apenas um usuário, a chamada `$post->user` retornará o objeto User com as informações do usuário ligado a aquela postagem.

Se em algum momento você quiser recuperar as postagens de um usuário, por exemplo do usuário 2, você pode seguir o pensamento do trecho de código abaixo:

```
1 ...
2 #No controller fazer a busca pelo usuário
3
4 $user = User::find(2);
5
6 #Acessar a ligação de usuário e postagens
7
8 $user->posts; //retornará as postagens do user
```

Se você quiser testar isso de forma rápida, o artisan disponibiliza o `tinker` uma smart cli que nos permite interagirmos com nossa aplicação em tempo de execução e fazermos alguns testes. Para acessarmos ele basta irmos ao nosso terminal e executarmos o comando abaixo:

```
1 php artisan tinker
```

```
blog(master) x: php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.22 - cli) by Justin Hileman
>>> 
```

Primeiramente vamos buscar pelo usuário, digite o trecho abaixo e dê um enter:

```
1 $user = \App\User::find(2);
```

Veja o resultado:

```
blog(master) x: php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.22 - cli) by Justin Hileman
>>> $user = \App\User::find(2);
=> App\User {#3029
    id: 2,
    name: "Reynold Schaefer",
    email: "alisa96@example.org",
    email_verified_at: "2019-09-23 20:34:18",
    created_at: "2019-09-23 20:34:18",
    updated_at: "2019-09-23 20:34:18",
}
```


Agora vamos acessar os posts deste usuário, execute o trecho abaixo e dê um enter:

```
1 $user->posts;
```

Veja o resultado:

```
>>> $user->posts;
=> Illuminate\Database\Eloquent\Collection {#3037
  all: [
    App\Post {#3038
      id: 8,
      user_id: 2,
      title: "autem non est eaque",
      description: "Fugit sunt ea architecto perferendis voluptate mollitia
sunt.",
      content: ""
        Nobis doloribus ut voluptas dolorum dolor. In deserunt voluptate est
totam repellendus et deleniti. Veniam totam eaque odio error ipsum vitae beata
e iusto. Optio dolor pariatur dignissimos sit deleniti ipsam delectus.\n
        \n
        Reprehenderit vero ut minima deserunt. Exercitationem odio sequi dig
nissimos labore excepturi sapiente voluptate velit.\n
        \n
        Dolorem voluptatem quas odit non omnis enim. Sequi labore animi qui
iste. Exercitationem sunt possimus dolor autem architecto consequatur.\n
        \n
        Molestias esse assumenda eum aut tempore iste. Quos ut veritatis eve
```

Veja acima um trecho da execução, como ele têm vários posts é retornado uma Collection com todos os posts deste usuário. Por meio deste wizard, que é o Tinker, agente consegue fazer diversas interações com nossa aplicação.

Esse objeto Collection(Illuminate\Database\Eloquent\Collection) contém diversos métodos que nos permitem interagirmos com a coleção em questão, como por exemplo contar quantos posts o usuário têm:

```
1 $user->posts->count();
```

Veja:

```
blog(master) x: php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.22 - cli) by Justin Hileman
>>> $user = \App\User::find(2);
=> App\User {#3029
  id: 2,
  name: "Reynold Schaefer",
  email: "alisa96@example.org",
  email_verified_at: "2019-09-23 20:34:18",
  created_at: "2019-09-23 20:34:18",
  updated_at: "2019-09-23 20:34:18",
}
>>> $user->posts->count();
=> 5
>>>
```

Agora vamos a inserção do autor para nossas postagens.

Inserindo Autor da Postagem

A minha intenção nesta inserção é termos o usuário logado e quando este criar a postagem nós pegamos a referência dele na sessão para adicionarmos no post mas como ainda não chegamos na parte de autenticação vou colocar direto no código a criação desta relação diretamente.

No próximo capítulo vamos conhecer a parte de autenticação e lá realizaremos essa melhoria. Sem mais delongas vamos a adição do autor do post no momento da criação da postagem.

Vamos ao nosso método store lá no PostController. Faça a seguinte alteração no código, o que está assim:


```

1 public function store(Request $request)
2 {
3     //Salvando com mass assignment
4     $data = $request->all();
5
6     $data['user_id'] = 1;
7     $data['is_active'] = true;
8
9     $post = new Post();
10
11     dd($post->create($data));
12 }

```

Ficará assim:

```

1 public function store(Request $request)
2 {
3     $data = $request->all();
4     $data['is_active'] = true;
5
6     $user = User::find(1);
7
8     //Continuamos a salvar com mass assignment mas por meio do usuário
9     dd($user->posts()->create($data));
10
11 }

```

Não esqueça de importar User:

```
1 use \App\User;
```

Perceba que agora nós inserimos uma nova postagem por meio da

ligação que temos com o usuário, como eu quero ter acesso aos métodos da ligação eu preciso chamar de fato o método `posts()` ao invés de chamar como atributo `posts`.

O Eloquent ao criar a postagem, irá adicionar a referência do usuário que buscamos por meio do método `find` automaticamente. Como a postagem pertence ao usuário, defini ele como prioridade quando busquei pelo mesmo e depois adicionei uma postagem ao seu conjunto de `posts`.

ManyToMany com Eloquent: Categorias e Posts

Agora neste ponto vamos começar a parte da relação de Muitos para Muitos, a relação aqui será entre `Posts` e `Categorias`. Primeiramente vamos criar o model `Category`(Categoria), suas migrations e todo o CRUD deste participante.

Podemos já começar criando nosso model com todo o aparato necessário de uma vez só, criar o model, a migration e o controller como recurso de uma vez só!

Execute na raiz do seu projeto o comando abaixo:

```
1 php artisan make:model Category -m -c -r
```

Considerações:

- `-m`: cria a migration deste model;
- `-c`: cria o controller deste model;
- `-r`: cria o controller como recurso para este model.

Veja o resultado:


```
blog: php artisan make:model Category -m -c -r
Model created successfully.
Created Migration: 2019_10_05_162546_create_categories_table
Controller created successfully.
blog: █
```

Obs.: O comando irá criar o controller na pasta de Controllers normalmente, apenas mova este controller para a pasta Admin e corrija o namespace, de namespace App\Http\Controllers; para namespace App\Http\Controllers\Admin; e adicione o import do controller base: use App\Http\Controllers\Controller;

Veja o controller na íntegra depois das observações acima:

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use App\Category;
6 use App\Http\Controllers\Controller;
7 use Illuminate\Http\Request;
8
9 class CategoryController extends Controller
10 {
11     /**
12      * Display a listing of the resource.
13      *
14      * @return \Illuminate\Http\Response
15      */
16     public function index()
17     {
18         //
19     }
20
```

```
21     /**
22      * Show the form for creating a new resource.
23      *
24      * @return \Illuminate\Http\Response
25      */
26     public function create()
27     {
28         //
29     }
30
31     /**
32      * Store a newly created resource in storage.
33      *
34      * @param \Illuminate\Http\Request $request
35      * @return \Illuminate\Http\Response
36      */
37     public function store(Request $request)
38     {
39         //
40     }
41
42     /**
43      * Display the specified resource.
44      *
45      * @param \App\Category $category
46      * @return \Illuminate\Http\Response
47      */
48     public function show(Category $category)
49     {
50         //
51     }
52
53     /**
```



```

54  * Show the form for editing the specified resource.
55  *
56  * @param \App\Category $category
57  * @return \Illuminate\Http\Response
58  */
59  public function edit(Category $category)
60  {
61      //
62  }
63
64  /**
65   * Update the specified resource in storage.
66   *
67   * @param \Illuminate\Http\Request $request
68   * @param \App\Category $category
69   * @return \Illuminate\Http\Response
70   */
71
72  public function update(Request $request, Category $category)
73  {
74      //
75  }
76
77  /**
78   * Remove the specified resource from storage.
79   *
80   * @param \App\Category $category
81   * @return \Illuminate\Http\Response
82   */
83  public function destroy(Category $category)
84  {
85      //

```

```
86 }
```

Por enquanto vamos deixá-lo assim, vamos dar uma atenção lá para nossa migration, acesse a pasta de migrations e abra o arquivo: `2019_10_05_162546_create_categories_table.php` (neste caso no momento da minha criação o nome é este, pra você pode estar diferente mas aí é só buscar o nome principal da migration).

Adicione os seguintes campos abaixo:

```

1 $table->string('name');
2 $table->string('description')->nullable();
3 $table->string('slug');

```

Como você pode ver o Laravel pegou o nome do nosso model e já preparou nossa migration para o plural, pela convenção já comentada aqui. Neste caso teremos a tabela `categories` com os campos:

- id;
- name;
- description;
- slug;
- created_at;
- updated_at;

Após as alterações que comentei anteriormente vamos criar uma nova migration, a migração para nossa tabela pivot ou intermediária para esta relação. Muitos para Muitos permite que uma postagem tenha várias categorias e uma categoria tenha várias postagens, pensando nisso precisamos de uma tabela intermediária para manter esta relação/ligação.

Execute em seu terminal o comando abaixo para criação da migração para esta tabela intermediária:


```
1 php artisan make:migration create_table_posts_categories
--create=posts_categories
```

Veja o conteúdo na íntegra da migração:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateTablePostsCategories extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16
17         Schema::create('posts_categories', function (Blueprint $table) {
18             $table->unsignedBigInteger('post_id');
19             $table->unsignedBigInteger('category_id');
20
21             $table->foreign('post_id')->references('id')->on('posts');
22
23             $table->foreign('category_id')->references('id')->on('categories');
24         });
25     }
26 }
```

```
25     /**
26      * Reverse the migrations.
27      *
28      * @return void
29      */
30     public function down()
31     {
32         Schema::dropIfExists('posts_categories');
33     }
34 }
```

Do método up remova o bigIncrements e o timestamps e adicione o conteúdo como acima. Nesta tabela precisamos apenas das referências, um para o post e o outro para a categoria, além das chaves estrangeiras referenciando cada tabela para o id de cada uma.

Agora podemos executar estas migrações na base de dados, vamos ao terminal executar o comando abaixo:

```
1 php artisan migrate
```

Veja o resultado:

```
blog: php artisan migrate
Migrating: 2019_10_05_162546_create_categories_table
Migrated: 2019_10_05_162546_create_categories_table (0.18 seconds)
Migrating: 2019_10_05_163809_create_table_posts_categories
Migrated: 2019_10_05_163809_create_table_posts_categories (0.17 seconds)
blog: █
```

Com as migrations definidas, como podemos mapear nossa relação do ponto de vista do model?

Existe no Eloquent o método para esta relação, o belongsToMany que estará nos dois models por conta da tabela intermediária. Então, vamos

adicionar o método abaixo dentro do nosso model Post:

```
1 public function categories()
2 {
3
4 return $this->belongsToMany(Category::class, 'posts_categories');
5 }
```

E no model Category adicione o método abaixo:

```
1 public function posts()
2 {
3
4 return $this->belongsToMany(Post::class, 'posts_categories');
5 }
```

O método belongsToMany nos permite o mapeamento na relação de muitos p/ muitos entre os dois models, como ambos apontam para a mesma tabela, o método de ambos os lados também é o mesmo.

Aqui vale um aprendizado e um adendo, o segundo parâmetro do método belongsToMany é o nome da tabela intermediária(posts_categories em nosso caso), mas por que eu preenchi este valor?

Eu preenchi valor deste parâmetro pois escolhi meu proprio nome de tabela intermediária que não é o mesmo que o Laravel iria tentar resolver de forma automática.

Mas Nanderson, como eu faço se eu quiser que o Laravel resolva automaticamente a tabela sem precisar do segundo parâmetro como tu fizestes?!

Aqui vai os aprendizados.

O Laravel, caso não exista o segundo parâmetro como fiz, vai buscar a tabela no banco respeitando o nome da tabela intermediária no singular e em ordem alfabética, ou seja, se temos posts e categories ele supõe que a tabela intermediária seja category_post, nomes no singular separados pelo _ e em ordem alfabética **c** antes do **p**. Tranquilo? Espero que sim!

Agora.

Antes de criarmos nosso post com suas categorias vamos criar rapidamente o CRUD de categorias com alguns detalhes a mais a serem adicionados também em nosso CRUD de posts melhorando assim nossa aplicação, Blog.

CRUD de Categorias

Vamos criar nosso CRUD com os controles de exceptions, bloco **try...**

catch, e com redirecionamentos e mensagens de execução para o usuário. Primeiramente vamos instalar um pacote para exibirmos mensagens de execução de cada processo para nosso usuário, execute o comando abaixo na raiz do projeto:

```
1 composer require laracasts/flash
```

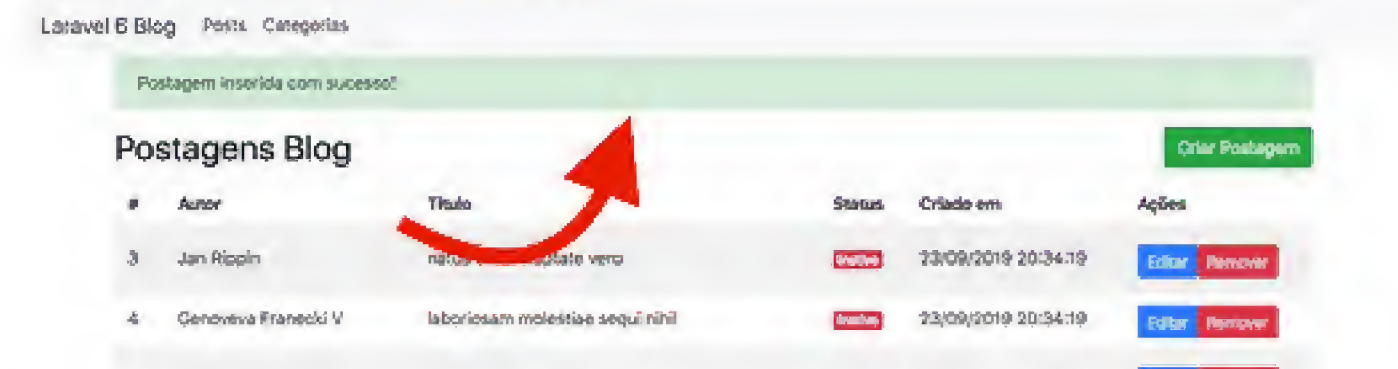
Este pacote nos permite de forma simples e rápida criarmos mensagens de execução para nossos usuários, mensagens que digam o que aconteceu no processo.

Por exemplo, após a inserção de uma postagem podemos jogar a mensagem via **laracasts/flash**, pro caso de sucesso da inserção, da seguinte maneira:

```
1 flash('Postagem criada com sucesso!')->success();
```

Isso produzirá na tela um sessão com a mensagem informada e com as

classes do Twitter Bootstrap: alert e alert-success, exibindo um alert verde pra nós. Como na imagem:



Agora vamos ao nosso controller, irei colocar método por método do controller de Categorias, o CategoryController, para vermos os pontos importantes sobre cada método. Ao final coloco ele na íntegra para conferência.

Vamos ao primeiro método, o método index e o construtor:

```

1 /**
2  * @var Category
3  */
4 private $category;
5
6 public function __construct(Category $category)
7 {
8     $this->category = $category;
9 }
10
11 /**
12  * Display a listing of the resource.
13  *
14  * @return \Illuminate\Http\Response
15  */
16 public function index()
```

```

17 {
18     $categories = $this->category->paginate(15);
19
20     return view('categories.index', compact('categories'));
21 }
22 }
```

O primeiro ponto importante é a utilização do nosso model como dependência do construtor da classe controller. Com isso teremos por meio do container de serviços do Laravel a instância do nosso model sempre que nosso controller for chamado. Isso simplifica muito as coisas e nos ajuda a mantermos nosso controller mais focado.

Falarei um pouco mais sobre o container de serviços mais a frente mas aqui já temos uma pequena pitada do que é possível fazer com esse cara!

Continuando para o método index, a diferença direta aqui é que ao invés de chamar o paginate assim `Category::paginate(15)` chamamos pelo atributo do controller que receberá a instância do nosso model.

O método a seguir é o create, que dispensa comentários até então, veja ele:

```

1 /**
2  * Show the form for creating a new resource.
3  *
4  * @return \Illuminate\Http\Response
5  */
6 public function create()
7 {
8     return view('categories.create');
9 }
```


Vamos continuando para o método store, onde de fato persistimos nossos dados, neste caso categoria. Veja ele e logo após os comentários:

```

1 /**
2  * Store a newly created resource in storage.
3  *
4  * @param \Illuminate\Http\Request $request
5  * @return \Illuminate\Http\Response
6  */
7 public function store(Request $request)
8 {
9     $data = $request->all();
10
11     try {
12         $category = $this->category->create($data);
13
14         flash('Categoria criada com sucesso!')->success();
15         return redirect()->route('categories.index');
16
17     } catch (\Exception $e) {
18         $message = 'Erro ao criar categoria!';
19
20         if(env('APP_DEBUG')) {
21             $message = $e->getMessage();
22         }
23
24         flash($message)->warning();
25         return redirect()->back();
26     }
27 }

```

Aqui mais uma referência ao nosso atributo que contém a instância do nosso model, usamos o mass assignment que já conhecemos. Agora temos mais alguns detalhes, primeiramente estamos controlando nossos

processos com os blocks **try...catch** que nos permite um maior controle caso as Exceptions com erros sejam lançadas.

Usamos também, tanto para as mensagens de execução de sucesso e de erro, o pacote que instalamos anteriormente, o laracasts/flash, onde definimos as mensagens e o tipo delas: se sucesso, `success()`, se error o `warning()`.

Logo após o set das mensagens de execução usamos o redirecionamento do Laravel por meio do helper `redirect` onde pro sucesso, redirecionamos o usuário para a tela principal de categorias dentro do admin, mas aqui por boa prática, redirecionamos pelo apelido da rota por isto chamo o método `route` que já usamos nas views!

```

1 flash('Categoria criada com sucesso!')->success();
2 return redirect()->route('categories.index');

```

Veja o bloco catch:

```

1 catch (\Exception $e) {
2     $message = 'Erro ao criar categoria!';
3
4     if(env('APP_DEBUG')) {
5         $message = $e->getMessage();
6     }
7
8     flash($message)->warning();
9     return redirect()->back();
10 }

```

Aqui faço um controle para só exibirmos as mensagens reais dos erros, se lá em nosso arquivo `.env` a variável `APP_DEBUG` estiver como verdadeira(true), ou seja, só veremos as mensagens de erro reais em ambiente de desenvolvimento, em produção teremos mensagens padrões como: "Erro ao criar categoria".

O método show têm um pequeno detalhe que preciso comentar, veja:

```
1 /**
2  * Display the specified resource.
3  *
4  * @param  \App\Category $category
5  * @return \Illuminate\Http\Response
6  */
7 public function show(Category $category)
8 {
9     return view('categories.edit', compact('category'));
10 }
```

Perceba de cara o parâmetro tipado do método show, tipado para aceitar apenas instâncias do nosso model Category. Um ponto importante aqui é que quando temos um parâmetro tipado assim, o Laravel automaticamente vai converter este parâmetro para um objeto populado para o id informado, por isso que não estou realizando um find ou findOrFail aqui, porque quando chegamos na execução deste método do controller já temos a instância do objeto Category populada com o id informado como comentei.

Então, só precisamos mandar para a view **edit** de categorias.

Quando nós geramos um controller como recurso, temos a geração também do método edit. Que serviria de fato para exibição do nosso formulário de edição.

Particularmente não teremos tela somente visualização dos dados de uma categoria em questão, para o qual o método show foi idealizado, por isso já exibo o formulário neste método e não no edit por escolha própria, geralmente o que adiciono no método edit é apenas fazer um redirecionamento, como fiz, veja abaixo:

```
1 /**
2  * Show the form for editing the specified resource.
3  *
4  * @param  \App\Category $category
5  * @return \Illuminate\Http\Response
6  */
7 public function edit(Category $category)
8 {
9
10     return redirect()->route('categories.show', ['category' => $category->id]);
11 }
```

Agora caímos para o método update, realmente aqui não temos muito a comentar uma vez que os detalhes já são conhecidos e a forma de atualização também:

```
1 /**
2  * Update the specified resource in storage.
3  *
4  * @param  \Illuminate\Http\Request $request
5  * @param  \App\Category $category
6  * @return \Illuminate\Http\Response
7  */
8
9 public function update(Request $request, Category $category)
10 {
11     $data = $request->all();
12
13     try {
14         $category->update($data);
15
16         flash('Categoria atualizada com sucesso!')->success();
17     }
18 }
```



```

16     return redirect()->route('categories.show', ['category' => $
category->id]);
17
18     } catch (\Exception $e) {
19         $message = 'Erro ao atualizar categoria!';
20
21         if(env('APP_DEBUG')) {
22             $message = $e->getMessage();
23         }
24
25         flash($message)->warning();
26         return redirect()->back();
27     }
28 }

```

A remoção/delete também segue o mesmo pensamento, veja:

```

1 /**
2  * Remove the specified resource from storage.
3  *
4  * @param \App\Category $category
5  * @return \Illuminate\Http\Response
6  */
7 public function destroy(Category $category)
8 {
9     try {
10         $category->delete();
11
12         flash('Categoria removida com sucesso!')->success();
13         return redirect()->route('categories.index');
14
15     } catch (\Exception $e) {
16         $message = 'Erro ao remover categoria!';
17

```

```

18         if(env('APP_DEBUG')) {
19             $message = $e->getMessage();
20         }
21
22         flash($message)->warning();
23         return redirect()->back();
24     }
25 }

```

Não esqueça dos parâmetros convertidos automaticamente para o model populado com base no id, resolvido dinamicamente pelo Laravel.

Ah, e não esqueça do atributo \$fillable lá no model Category, para o mass assignment:

```

1 protected $fillable = [
2     'name',
3     'description',
4     'slug'
5 ];

```

Veja o controller CategoryController na íntegra:

```

1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use App\Category;
6 use App\Http\Controllers\Controller;
7 use Illuminate\Http\Request;
8
9 class CategoryController extends Controller

```



```

10 {
11     /**
12      * @var Category
13      */
14     private $category;
15
16     public function __construct(Category $category)
17     {
18         $this->category = $category;
19     }
20
21     /**
22      * Display a listing of the resource.
23      *
24      * @return \Illuminate\Http\Response
25      */
26     public function index()
27     {
28         $categories = $this->category->paginate(15);
29
30         return view('categories.index', compact('categories'));
31     }
32
33     /**
34      * Show the form for creating a new resource.
35      *
36      * @return \Illuminate\Http\Response
37      */
38
39     public function create()
40     {
41         return view('categories.create');
42     }

```

```

43
44     /**
45      * Store a newly created resource in storage.
46      *
47      * @param \Illuminate\Http\Request $request
48      * @return \Illuminate\Http\Response
49      */
50     public function store(Request $request)
51     {
52         $data = $request->all();
53
54         try {
55             $category = $this->category->create($data);
56
57             flash('Categoria criada com
58             sucesso!')->success();
59             return redirect()->route('categories.index');
60         } catch (\Exception $e) {
61             $message = 'Erro ao criar categoria!';
62
63             if(env('APP_DEBUG')) {
64                 $message = $e->getMessage();
65             }
66
67             flash($message)->warning();
68             return redirect()->back();
69         }
70     }
71
72     /**
73      * Display the specified resource.
74      *
75      * @param \App\Category $category

```



```

76     * @return \Illuminate\Http\Response
77     */
78     public function show(Category $category)
79     {
80
81         return view('categories.edit', compact('category'));
82     }
83
84     /**
85      * Show the form for editing the specified resource.
86      *
87      * @param \App\Category $category
88      * @return \Illuminate\Http\Response
89      */
90     public function edit(Category $category)
91     {
92
93         return redirect()->route('categories.show', ['category' => $
category->id]);
94     }
95
96     /**
97      * Update the specified resource in storage.
98      *
99      * @param \Illuminate\Http\Request $request
100     * @param \App\Category $category
101     * @return \Illuminate\Http\Response
102     */
103     public function update(Request $request, Category $category)
104     {

```

```

105         try {
106             $category->update($data);
107
108             flash('Categoria atualizada com
sucesso!')->success();
109
110             return redirect()->route('categories.show', ['category' => $
category->id]);
111         } catch (\Exception $e) {
112             $message = 'Erro ao atualizar categoria!';
113
114             if(env('APP_DEBUG')) {
115                 $message = $e->getMessage();
116             }
117
118             flash($message)->warning();
119             return redirect()->back();
120         }
121     }
122
123     /**
124      * Remove the specified resource from storage.
125      *
126      * @param \App\Category $category
127      * @return \Illuminate\Http\Response
128      */
129     public function destroy(Category $category)
130     {
131         try {
132             $category->delete();
133
134             flash('Categoria removida com
sucesso!')->success();

```



```

135         return redirect()->route('categories.index');
136
137     } catch (\Exception $e) {
138         $message = 'Erro ao remover categoria!';
139
140         if(env('APP_DEBUG')) {
141             $message = $e->getMessage();
142         }
143
144         flash($message)->warning();
145         return redirect()->back();
146     }
147 }
148 }

```

Agora, vamos as views.

As views também não têm novidades, mas antes, crie a pasta `categories` dentro da pasta `views` e dentro os arquivos:

- `index.blade.php`;
- `create.blade.php`;
- `edit.blade.php`.

Com os conteúdos:

index.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <div class="row">
5         <div class="col-sm-12">
6             <a href="
{{route('categories.create')}}" class="btn btn-success float-
ri\

```

```

7 ght">Criar Categoria</a>
8         <h2>Categorias Blog</h2>
9         <div class="clearfix"></div>
10     </div>
11 </div>
12 <table class="table table-striped">
13     <thead>
14         <tr>
15             <th>#</th>
16             <th>Nome</th>
17             <th>Criado em</th>
18             <th>Ações</th>
19         </tr>
20     </thead>
21     <tbody>
22         @foreach($categories as $category)
23             <tr>
24                 <td>{{ $category->id }}</td>
25                 <td>{{ $category->name }}</td>
26
27                 <td>{{date('d/m/Y H:i:s', strtotime($category->created_at))}}
28                 </td>
29                 <td>
30                     <div class="btn-group">
31                         <a href="{{route('categories.show',
32                             ['category' => $category\
33                             ->id])}}" class="btn btn-sm btn-primary">
34                             Editar
35                         </a>
36                         <form action="
37                             {{route('categories.destroy', ['category' => $
38                             category->id])}}" method="post">

```



```

35         @csrf
36         @method('DELETE')
37         <button type="submit" class="btn btn-sm btn-danger">Remo\
38 ver</button>
39     </form>
40 </div>
41 </td>
42 </tr>
43 @empty
44 <tr>
45     <td colspan="4">Nada encontrado!</td>
46 </tr>
47 @endforelse
48 </tbody>
49 </table>
50
51 {{ $categories->links() }}
52 @endsection

```

create.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="
5 {{route('categories.store')}}> method="post">
6
7         @csrf
8
9         <div class="form-group">
10             <label>Nome</label>
11             <input type="text" name="name" class="form-
12 control" value="{{old('name')\

```

```

11 }}">
12     </div>
13
14     <div class="form-group">
15         <label>Descrição</label>
16
17         <input type="text" name="description" class="form-
18 control" value="{{old(\
19 'description')}}">
20     </div>
21
22     <div class="form-group">
23         <label>Slug</label>
24         <input type="text" name="slug" class="form-
25 control" value="{{old('slug')\
26 }}">
27     </div>
28
29     <button class="btn btn-lg btn-
30 success">Criar Categoria</button>
31 </form>
32 @endsection

```

edit.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{route('categories.update', ['category'
5 => $category->id])}}> met\
6 hod="post">
7
8         @csrf
9         @method("PUT")

```



```

9
10     <div class="form-group">
11         <label>Nome</label>
12         <input type="text" name="name" class="form-
13 control" value="{{ $category->\
14 name }}">
15     </div>
16     <div class="form-group">
17         <label>Descrição</label>
18         <input type="text" name="description" class="form-
19 control" value="{{ $cat\
20 egory->description }}">
21     </div>
22     <div class="form-group">
23         <label>Slug</label>
24         <input type="text" name="slug" class="form-
25 control" value="{{ $category->\
26 slug }}">
27     </div>
28     <button class="btn btn-lg btn-
29 success">Atualizar Categoria</button>
30 </form>
31 @endsection

```

Criada as views de categorias, precisamos adicionar lá no nosso layout (app.blade.php) a possibilidade de exibição de nossas mensagens do pacote laracasts/flash. Dentro do nosso bloco onde se encontra o `yield('content')`, adicione acima dele o include abaixo:

```
1 @include("flash::message")
```

Veja como ficou o trecho do yield:

```

1 ...
2 <div class="container">
3     @include("flash::message")
4     @yield('content')
5 </div>
6 ...

```

Ah e não esqueça de link no menu do layout.blade.php o menu de categorias, veja o trecho adicionado logo após o link de posts:

```

1 <li class="nav-item active">
2     <a class="nav-
3 link" href="{{ route('categories.index') }}">Categorias</a>
4 </li>

```

Esta inclusão exibirá os alertas com as classes do twitter bootstrap conforme o método escolhido, na cor verde para a chamada do `->success()` na cor amarela para a chamada `->warning()`.

Agora que nosso CRUD de categorias está pronto, faça alguns testes e crie algumas categorias para termos insumo no momento da inclusão das categorias para uma postagem.

Alterações em PostController

Abaixo, deixo nosso controller PostController alterado com os pensamentos feitos no controller de categorias. Como não criamos ele diretamente como recursos pelo terminal, ele vai está sem alguns comentários como vimos no CategoryController mas isso são detalhes.

Veja ele na íntegra:

```
1 <?php
```



```

2
3 namespace App\Http\Controllers\Admin;
4
5 use App\User;
6 use Illuminate\Http\Request;
7 use App\Http\Controllers\Controller;
8 use App\Post;
9
10 class PostController extends Controller
11 {
12     /**
13      * @var Post
14      */
15     private $post;
16
17     public function __construct(Post $post)
18     {
19         $this->post = $post;
20     }
21
22     public function index()
23     {
24         $posts = $this->post->paginate(15);
25
26         return view('posts.index', compact('posts'));
27     }
28
29     public function create()
30     {
31         return view('posts.create');
32     }
33
34     public function store(Request $request)
35     {

```

```

36         $data = $request->all();
37         try{
38             $data['is_active'] = true;
39
40             $user = User::find(1);
41             $user->posts()->create($data);
42
43             flash('Postagem inserida com
sucesso!')->success();
44             return redirect()->route('posts.index');
45
46         } catch (\Exception $e) {
47             $message = 'Erro ao remover categoria!';
48
49             if(env('APP_DEBUG')) {
50                 $message = $e->getMessage();
51             }
52
53             flash($message)->warning();
54             return redirect()->back();
55         }
56     }
57
58     public function show(Post $post)
59     {
60         return view('posts.edit', compact('post'));
61     }
62
63
64     public function update(Post $post, Request $request)
65     {
66         $data = $request->all();
67
68         try{
69             $post->update($data);

```



```

36     $data = $request->all();
37     try{
38         $data['is_active'] = true;
39
40         $user = User::find(1);
41         $user->posts()->create($data);
42
43         flash('Postagem inserida com
sucesso!')->success();
44         return redirect()->route('posts.index');
45     } catch (\Exception $e) {
46         $message = 'Erro ao remover categoria!';
47
48         if(env('APP_DEBUG')) {
49             $message = $e->getMessage();
50         }
51
52         flash($message)->warning();
53         return redirect()->back();
54     }
55 }
56
57 public function show(Post $post)
58 {
59     return view('posts.edit', compact('post'));
60 }
61
62
63
64 public function update(Post $post, Request $request)
65 {
66     $data = $request->all();
67
68     try{
69         $post->update($data);

```

```

70
71         flash('Postagem atualizada com
sucesso!')->success();
72
73         return redirect()->route('posts.show', ['post' => $post->id]
);
74     } catch (\Exception $e) {
75         $message = 'Erro ao remover categoria!';
76
77         if(env('APP_DEBUG')) {
78             $message = $e->getMessage();
79         }
80
81         flash($message)->warning();
82         return redirect()->back();
83     }
84 }
85
86 public function destroy(Post $post)
87 {
88     try {
89         $post->delete($post);
90
91         flash('Postagem removida com
sucesso!')->success();
92         return redirect()->route('posts.index');
93     } catch (\Exception $e) {
94         $message = 'Erro ao remover categoria!';
95
96         if(env('APP_DEBUG')) {
97             $message = $e->getMessage();
98         }
99     }
100 }

```



```

101         flash($message)->warning();
102         return redirect()->back();
103     }
104 }
105 }

```

Aplicada as melhorias no controller `PostController` vamos para à alterações necessárias para inserção da relação muitos para muitos entre Posts e Categorias.

Inserindo Muito para Muitos (Post x Category)

Para inserção de muitos para muitos, nós temos 3 métodos principais, o método `attach`, o `detach` e o `sync` que particularmente gosto de utilizar. Utilizarei aqui o `sync` mas darei uma rápida amostra do que os 2 outros fazem.

O método `attach` recebe um array com os ids a serem incluídos na referência, se estivermos salvando do ponto de vista de Post os ids serão de categorias e vice-versa. Veja o exemplo:

```
1 $post->categories()->attach([1,2]);
```

Com o `attach` acima estou adicionando para uma postagem salva recentemente duas categorias, de id 1 e 2 respectivamente. Se em uma tela de edição eu quiser remover a categoria 1 deste post, eu posso utilizar o `detach`, como abaixo:

```
1 $post->categories()->detach([1]);
```

Agora como o `sync` funciona, ele é bem simples. Basicamente ele irá sincronizar as referências da ligação, por exemplo:

```
1 $post->categories()->sync([1,2]);
```

Acima o `sync` irá inserir as duas referências caso elas não existam na ligação, dentro da tabela intermediária. Uma vez salva, se eu utilizar o `sync` novamente da maneira abaixo:

```
1 $post->categories()->sync([1]);
```

Ele irá remover os ids que não estiverem no array informado a ele, neste caso, a categoria de id 2 será removida da ligação.

Alterando Posts para Inserção N:N

Vamos adicionar a possibilidade de adição de categorias em nossas postagens. O primeiro passo é mandarmos para as views de criação e edição nossas categorias.

Veja como ficou os métodos agora `create` e `update` lá no `PostController`:

create

```

1 public function create()
2 {
3     $categories = \App\Category::all(['id', 'name']);
4
5     return view('posts.create', compact('categories'));
6 }

```

edit

```

1 public function show(Post $post)
2 {
3     $categories = \App\Category::all(['id', 'name']);
4
5
6     return view('posts.edit', compact('post', 'categories'));
7 }

```


Agora mandamos para cada view as categorias existentes em nossa base, neste caso passei um array para o método `all` que me retornará apenas o id e o nome de cada categoria, que é o que necessitamos pra este momento.

Agora precisamos exibir essas categorias em nossos formulários. Adicione o trecho abaixo em ambos os formulários:

```
1 <div class="form-group">
2   <label>Categorias</label>
3   <div class="row">
4     @foreach($categories as $c)
5       <div class="col-2 checkbox">
6         <label>
7
8         <input type="checkbox" name="categories[]" value="
9         {{$c->id}}"> {\
10        {{$c->name}}
11      </label>
12    </div>
13  @endforeach
14 </div>
```

Este trecho exibirá diversas checkboxes com as categorias e seus ids como valores para cada checkbox. Perceba um detalhe importante, o checkbox nos permite marcarmos quantos itens quisermos, neste caso como preciso mandar um array de ids (das categorias) para sincronizar, usei a notação no nome do campo com colchetes ([]) desta maneira: `categories[]`. Assim enviaremos na requisição um array com os ids selecionados no campo `categories`, vindo dos checkboxes.

Uma vez feita esta alteração, vamos adicionar a possibilidade de save da relação, adicionando assim as categorias do post. Para isto, basta

adicionarmos logo após a linha de criação e atualização dentro dos seus métodos (`store`, `update`), a chamada do `sync` como vemos abaixo:

```
1 $post->categories()->sync($data['categories']);
```

Veja o método `create` e o `update` na íntegra, pós alteração:

`create`

```
1 public function store(Request $request)
2 {
3     $data = $request->all();
4     try{
5         $data['is_active'] = true;
6
7         $user = User::find(1);
8
9         $post = $user->posts()->create($data); //Retornará o Post in
10        serido, atribuímos \
11        ele a variável post para usarmos abaixo no sync
12
13        $post->categories()->sync($data['categories']); //aqui
14
15        flash('Postagem inserida com sucesso!')->success();
16        return redirect()->route('posts.index');
17
18    } catch (\Exception $e) {
19        $message = 'Erro ao remover categoria!';
20
21        if(env('APP_DEBUG')) {
22            $message = $e->getMessage();
23        }
```



```

24
25     flash($message)->warning();
26     return redirect()->back();
27 }
28 }

update

1 public function update(Post $post, Request $request)
2 {
3     $data = $request->all();
4
5     try{
6         $post->update($data);
7         $post->categories()->sync($data['categories']);
//aqui
8
9         flash('Postagem atualizada com
sucesso!')->success();
10        return redirect()->route('posts.show', ['post' =>
$post->id]);
11
12    } catch (\Exception $e) {
13        $message = 'Erro ao remover categoria!';
14
15        if(env('APP_DEBUG')) {
16            $message = $e->getMessage();
17        }
18
19        flash($message)->warning();
20        return redirect()->back();
21    }
22 }

```

Desta forma já adicionamos a possibilidade de inserção e edição das

categorias para as postagens. Agora precisamos selecionar as categorias da postagem a ser edita, na tela de edição e sabermos quais as categorias do post quando entrarmos nesta tela.

Como temos uma relação de muitos para muitos, se acessarmos por exemplo:

```
1 $post->categories;
```

Teremos uma Collection, como vimos anteriormente, com as categorias existentes para a postagem em questão. Essa Collection nos permite verificarmos se determinado Objeto existe dentro da coleção, por meio do método `contains`. O que precisamos é marcar com `checked` cada input que bater com a categoria existente na relação, dentro do loop de categorias teremos cada categorias em questão, o que precisamos é passar para o método `contains` cada linha para que o Laravel verifique se aquela categoria em questão está na relação!

Vamos a alteração para entederemos melhor, na view de edição (`edit.blade.php`) de posts o que está assim, dentro do loop de categorias:

```
1 <input type="checkbox" name="categories[]" value="
{{$c->id}}"> {{$c->name}}
```

Ficará assim:

```
1 <input type="checkbox" name="categories[]" value="
{{$c->id}}"
2
3 @if($post->categories->contains($c)) checked @endif
4
5 > {{$c->name}}
```

Como estamos fazendo o laço nas categorias, em cada linha teremos o

objeto Category populado com a categoria em questão. Com isso podemos verificar se na relação, do model, entre categorias e posts, existe aquele objeto disponível usando o método `contains`.

Existindo, o `contains` retornará verdadeiro, e aí nós adicionamos a propriedade `checked` no input. Selecione uma categoria e atualize a postagem de sua escolha e verifique que agora a(s) categoria(s) pertencente a postagem estará ou estarão checkada(s).

127.0.0.1:8000/admin/posts/3

notas para voluptate vero

Descrição

Optio illo nihil repellendus a adipisci.

Conteúdo

Nihil pequi vel dolores voluptate. Quotibus ea omnis in velit nulla. Non atque laudantium quidem eos.

Dulpa vires ut dolorem molestiae ex. Voluptatem veritatis eos quia repudiandae aut tenetur aut. Est necessitatibus corrupti explicabo officia ea accusamus enim.

Quia sit perferendis nemo nesciunt molestias officis qui. Repellendus qui aut ut hic asperiores. Autem voluptatem consequatur reprehenderit eveniet nulla nemo consequuntur.

Esse quod optio omnia molestias distinctio consequuntur. Debitis saepe debitis debitis incidunt quo. Assumenda ducimus at voluptas atque non facilis. Malores nihil voluptatibus non cum ratione eaque.

Slug

excepturi-acusamus-laborum-architecto-voluptate-mollitia-iste

Categorias

☐ Games ☒ Notícias

Atualizar Postagem

Remover Post

Na tela de inserção só precisamos selecionar as categorias que desejarmos que elas já serão salvas pelo `sync` e isso já fizemos. Faça seus testes!

Com isso chegamos ao ponto final proposto aqui para este capítulo.

Conclusões

O conteúdo abordado aqui foi bem denso, e nos requereu bastante alterações, entretanto, agora temos um projeto bem mais robusto e mais organizado do ponto de vista de execuções, controle de exceções, relação e módulos e ainda a exibição das mensagens pro usuário sobre os processos ocorridos.

No próximo capítulo vamos conhecer a camada de autenticação e como podemos privar o acesso aos nossos CRUDs a apenas usuários autenticados.

Até lá!

Autenticação

Neste capítulo vamos criar nossa autenticação para acesso ao nosso painel administrativo. O mais interessante é que o Laravel já vem com praticamente tudo pronto para criarmos esta autenticação de forma rápida e produtiva.

Esta dinamização e coisas prontas são o essencial para que possamos levantar nosso controle para o acesso ao nosso painel.

Então vamos lá, vamos implementar a autenticação e conhecer alguns conceitos dentro do framework, além de criarmos o perfil do usuário logado e conhecer de quebra a relação 1 para 1 (1:1).

Começando com a geração da autenticação

Primeiramente vamos instalar o pacote UI, o Laravel UI. Este pacote vai adicionar para nós a possibilidade de manipulação dos assets do frontend e também vai nos permitir a geração das views de login, resete de senha, registro de usuário e adicionará as rotas necessárias para este processo via comandos no artisan.

Para instalarmos basta executarmos na raiz do projeto o comando abaixo:

```
1 composer require laravel/ui
```

```
blog(master): composer require laravel/ui
Using version ^1.1 for laravel/ui
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing laravel/ui (v1.1.1): Loading from cache
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: facade/ignition
Discovered Package: fideloper/proxy
Discovered Package: laracasts/flash
Discovered Package: laravel/tinker
Discovered Package: laravel/ui
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
blog(master) x: █
```

Com pacote instalado poderemos executar o comando abaixo, para gerarmos nossas views de autenticação, bem como suas rotas e ainda ambientarmos nossos assets frontend com o Twitter Bootstrap. Ainda na raiz do seu projeto execute o comando abaixo:

```
1 php artisan ui bootstrap --auth
```



```
blog(master) X:
php artisan ui bootstrap --auth
Bootstrap scaffolding installed successfully.
Please run "npm install && npm run dev" to compile your fresh scaffolding.

The [layouts/app.blade.php] view already exists. Do you want to replace it? (yes/no) [no]:
> no

Authentication scaffolding generated successfully.
blog(master) X: █
```

O comando acima gera para nós o `app.blade.php` dentro de uma pasta `layouts`, em nossa pasta de `views` mas como já temos esse arquivo ele vai perguntar se eu quero sobrescrever, coloquei que não: `no`. Após isso ele continua a geração das `views` da autenticação, que fica na pasta `views` dentro da pasta `auth` e adiciona também lá no nosso arquivo de rotas `web.php` as rotas da autenticação por meio do trecho:

```
1 \Auth::routes();
```

Ele vai adicionar também o trecho abaixo e criar o `controller` `HomeController`:

```
1 Route::get('/home', 'HomeController@index')->name('home');
```

Que você pode descartar, ambos, por hora. Eles não vão ser necessários para nós, tanto o trecho da rota `home` como o `HomeController`.

O comando gerou o pontos acima por conta do parâmetro `--auth` e o

tipo passando para o comando `ui`, o `bootstrap`, fez com que o `Laravel` criasse as configurações necessárias para que possamos usar o `bootstrap` a partir dos `assets` dentro do projeto.

Você pode dar uma olhada nos arquivos:

- `resources/css/app.scss`;
- `resources/sass/_variables.scss`.

Também foi adicionado as dependências dentro do `package.json`:

- `bootstrap: ^4.0.0`;
- `jquery: ^3.2`;
- `popper.js: ^1.12`.

Para podermos usar estes `assets` no projeto, precisamos instalar essas dependências via `npm` (`Node Package Manager`). Neste caso será necessário ter o `Node.js` em sua máquina junto com o `NPM` (em alguns `Linux` ele vêm separado).

Tendo ambos, execute na raiz do projeto o comando abaixo:

```
1 npm i
```

O `i` aqui é de `install`.

```
blog(master) X: npm i
(⊙) i: fetchMetadata: sill pacote range manifest for node-notifier
```

Após o término, precisamos gerar um `build` com todos os `assets`, tanto o `css` quanto o `js`, que serão gerados dentro das pastas `css` e `js` na pasta `public` do projeto. Os arquivos gerados e prontos para uso são:

- `public/js/app.js`;
- `public/css/app.css`.

Para termos estes dois arquivos gerados, precisamos rodar o seguinte

comando na raiz do projeto:

```
1 npm run dev
```

Este comando vai gerar os builds dos assets para podermos utilizar os pacotes instalados via npm.

```
blog(master) X: npm run dev
> @ dev /Users/NandoKstroNet/books/blog
> npm run development

> @ development /Users/NandoKstroNet/books/blog
> cross-env NODE_ENV=development node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js
```

O sucesso da geração destes builds você pode ver abaixo:

```
DONE Compiled successfully in 18622ms 2:51:19 PM

  Asset      Size  Chunks             Chunk Names
/css/app.css 191 KiB /js/app [emitted] /js/app
/js/app.js   1.06 MiB /js/app [emitted] /js/app
```

O comando acima lerá os arquivos `resources/js/app.js` e `resources/css/app.scss` e criará o build para cada um dentro do `public` para podermos utilizar em nosso projeto. Todo esse processo que feito pelo pacote `larave/mix` que usa o WebPack para realizar dos

builds do frontend.

As rotas de autenticação

Um dos pontos que comentei foi a adição das rotas para autenticação por meio da chamada, lá no arquivo `web.php`, do trecho abaixo:

```
1 \Auth::routes();
```

As rotas adicionadas, por meio da chamada acima, são:

- `/login`: GET exibe o form e POST para submissão do login;
- `/logout`: POST para encerrar sessão;
- `/register`: Registro do Usuário (GET e POST);
- `/password/email`: POST envia o email para reset de senha;
- `/password/reset`: POST atualiza a senha;
- `/password/reset/{token}`: GET tela para atualização da senha e verificação do token de reset de senha.

Por exemplos se iniciarmos nosso servidor e acessarmos <http://127.0.0.1:8000/login> teremos o resultado abaixo:



Teremos nossa tela de login que foi gerada anteriormente.

Agora temos um detalhe, nossas rotas ainda não estão sobre a autenticação. Se tentarmos acessá-las vamos conseguir mesmo não estando logados. O que precisamos fazer para colocar nossa rota sob autenticação?

Neste caso precisamos usar o middleware `auth` que faz o controle nas rotas que usarem ele, e permitirá que somente usuários autenticados tenham acesso as mesmas. Tranquilo! Mas o que são middlewares?

Vamos lá!

Middlewares

Antes de usarmos o middleware `auth` para bloquear o acesso ao nosso painel vamos entender o conceito de middlewares.

Middlewares são aplicações de controle que são executadas entre a requisição do usuário e o código específico para aquela rota. Dentro do Laravel nós temos middlewares para a aplicação como um todo e middlewares específicos para rotas.

Como middlewares estão entre a requisição do usuário e a execução para a rota solicitada podemos realizar diversos controles como por exemplo verificar se o usuário está logado, via middleware `auth` ou até mesmo validar papéis do usuário em relação à aquele acesso. As possibilidades com middlewares são imensas e vai da necessidade de cada aplicação.

Usando o middleware AUTH

Para colocarmos nossas rotas do admin sob autenticação, precisamos fazer algumas alterações em nosso arquivo de rotas web. O trecho abaixo, que está assim:

```
1
Route::prefix('admin')->namespace('Admin')->group(function(){
2
3     Route::resource('posts', 'PostController');
4     Route::resource('categories', 'CategoryController');
5
6 });
```

Passará a ser utilizado da seguinte maneira:

```
1 Route::group(['middleware' => ['auth']], function(){
2
3
4     Route::prefix('admin')->namespace('Admin')->group(function()
5
6         Route::resource('posts', 'PostController');
7
8         Route::resource('categories', 'CategoryController');
9
10    });
```

Agora aplicamos o middleware `auth` para o nosso grupo de rotas do admin existentes. Perceba a chamada do método `group` e a utilização do primeiro parâmetro com o array passado, informando nossa chave `middleware` e dentro o array de middlewares para estas rotas, as rotas do admin.

Se você tentar acessar por exemplo: **`http://127.0.0.1/admin/posts`**, você será redirecionado para a tela de login, a simples adição deste middleware já nos permite este controle. Agora você precisa se autenticar no form de login para ter acesso ao administrativo.

PS.: Para testar você pode ir no seu banco e pegar algum usuário para teste e usar a senha lá da factory: password, combinada com o email que você escolheu.

Após o login com sucesso se você foi redirecionado para a rota /home e recebeu um 404, isso é porque: Primeiro, nós removemos a rota /home bem como seu controller e por padrão a rota /home é para onde o Laravel aponta o usuário depois de logado, registrado e atualizado a senha.

Podemos modificar este comportamento. Vamos realizar esta pequena alteração, acesse seu arquivo LoginController.php lá na pasta app/Http/Controllers/Auth e modifique o trecho abaixo:

```
1 /**
2  * Where to redirect users after login.
3  *
4  * @var string
5  */
6 protected $redirectTo = '/home';
```

para:

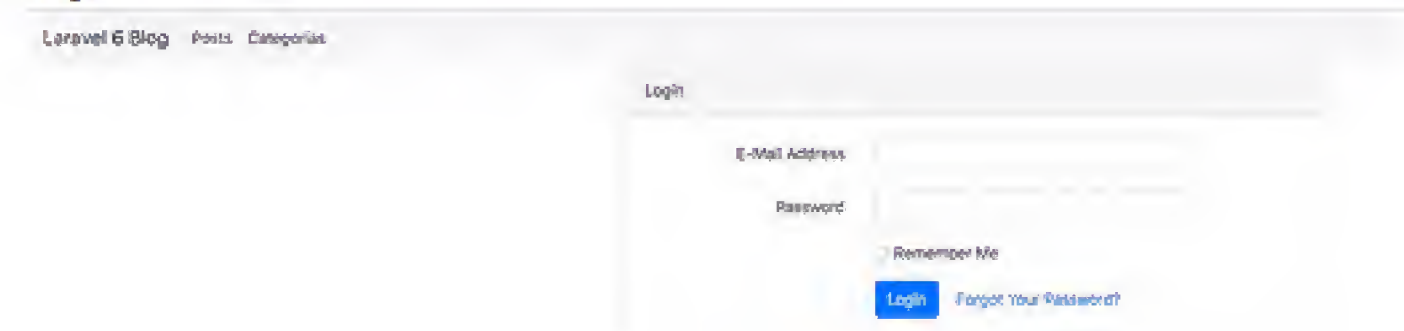
```
1 /**
2  * Where to redirect users after login.
3  *
4  * @var string
5  */
6 protected $redirectTo = '/admin/posts';
```

Esta variável \$redirectTo pode ser modificado ainda, nos controllers responsáveis por cada operação referentes a autenticação como: RegisterController, ResetPasswordController e ainda VerificationController na mesma pasta do LoginController.

Blade, controles para autenticação

Se você analisou a tela de login vai perceber que nossos menus Posts e Categorias estão sendo exibidos no menu mesmo não estando autenticados.

Veja:



Como fazer para exibirmos estes menus apenas quando estivermos logados? Simples, o blade têm duas diretivas para isto, uma para verificação de usuários autenticado:

```
1 @auth
2
3 @endauth
```

e uma para verificação de usuários não autenticados ou anônimos ou mesmo convidados:

```
1 @guest
2
3 @endguest
```

Ambas podem ter um senão, caso você necessite de um resultado default, se o usuário não está autenticado ou caso ele esteja autenticado. Veja:


```

1 @auth
2     Usuário autenticado...
3 @else
4     Usuário não autenticado...
5 @endauth

```

```

1 @guest
2     Usuário não autenticado...
3 @else
4     Usuário autenticado...
5 @endguest

```

Vamos usar o @auth em nosso menu, envolva o menu lá no app.blade.php como abaixo:

```

1 @auth
2     <ul class="navbar-nav">
3         <li class="nav-item active">
4             <a class="nav-link" href="{{
route('posts.index')}}">Posts</a>
5         </li>
6         <li class="nav-item active">
7             <a class="nav-link" href="{{
route('categories.index')}}">Categorias</a>
8         </li>
9     </ul>
10 @endauth

```

Se voltar a tela de login e atualizar, você perceberá que nosso menu já não aparece mais entretanto se logarmos ele estará lá!

Recuperando o usuário autenticado

Lembra, lá no controller, das postagens onde nós buscamos o usuário e

criamos a referência dele diretamente e que depois iríamos substituir pelo usuário logado? Então, vamos realizar esta alteração e verificar como podemos ter acesso aos dados do usuário que está autenticado/logado no momento.

Na verdade isso é tão simples quanto adicionar o controle do tópico anterior. Primeiramente vamos mão na massa, lá no PostController a linha no método store que está assim:

```
1 $user = User::find(1);
```

Ficará assim:

```
1 $user = auth()->user();
```

A função helper auth() retornará uma instância do gerenciador de sessão do pacote de auth do Laravel, e por meio do método user() teremos acesso ao objeto do usuário logado, onde poderemos acessar diversas informações do mesmo bem como as ligações e relações via model deste. Essa simples alteração já adicionará os posts criados para o usuário que está autenticado.

Simple, rápido e direto!

Caso queira exibir lá na view do layout o nome do usuário autenticado é bem simples também, veja o trecho e depois a view completa e atualizada:

resources/views/layoutus/app.blade.php:

Trecho adicionado antes do fechamento da tag nav:

```

1 @auth
2     <div class="float-right">
3         <strong>{{auth()->user()->name}}</strong>

```



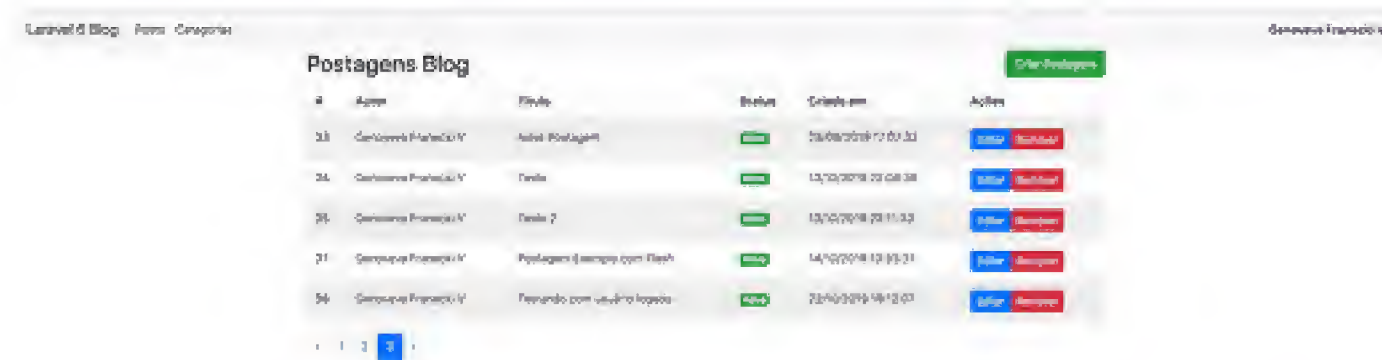
```
4     </div>
5 @endauth
```

Veja a view completa:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no,
7         initial-scale=1.0, maximum-
8         scale=1.0, minimum-scale=1.0">
9     <meta http-equiv="X-UA-Compatible" content="ie=edge">
10    <title>Gerenciador de Posts</title>
11    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.
12    com/bootstrap/4.3.1/\
13    css/bootstrap.min.css">
14    </head>
15    <body>
16    <nav class="navbar navbar-expand-lg navbar-light bg-
17    light">
18        <a class="navbar-brand" href="/">Laravel 6 Blog</a>
19        <button class="navbar-toggler" type="button" data-
20        toggle="collapse" data-target="#navbarNavDropdown" aria-
21        controls="navbarNavDropdown" aria-expanded="false" ari\
22        a-label="Toggle navigation">
23            <span class="navbar-toggler-icon"></span>
24        </button>
25        <div class="collapse navbar-
26        collapse" id="navbarNavDropdown">
```

```
27            @auth
28                <ul class="navbar-nav">
29                    <li class="nav-item active">
30                        <a class="nav-link" href="{{
31                        route('posts.index') }}">Posts<\
32                        /a>
33                    </li>
34                    <li class="nav-item active">
35                        <a class="nav-link" href="{{
36                        route('categories.index') }}">C\
37                        ategorias</a>
38                    </li>
39                </ul>
40            @endauth
41        </div>
42
43        @auth
44            <div class="float-right">
45                <strong>{{auth()->user()->name}}</strong>
46            </div>
47        @endauth
48    </nav>
49    <div class="container">
50        @include("flash::message")
51        @yield('content')
52    </div>
53 </body>
54 </html>
```

Imagem:



Feita estas alterações vamos ao perfil do autor e conhecer a ligação de Um para Um (OneToOne) ou 1:1. Vamos lá!

Relação 1:1 (Autor e Perfil)

Primeramente vamos criar nosso model e todo o aparato de uma vez só!

Execute o comando abaixo em seu terminal e na raiz do projeto:

```
1 php artisan make:model Profile -m -c
```

Não esqueça de mover seu controller para a pasta Admin e corrigir o namespace.

O conteúdo do controller gerado está abaixo:

```
1 <?php
2
3 namespace App\Http\Controllers\Admin;
4
5 use Illuminate\Http\Request;
6 use App\Http\Controllers\Controller;
7
8
9 class ProfileController extends Controller
10 {
11     //
12 }
```

Não gerei ele como recurso pois vamos precisar só do método para exibição do form do perfil e um para atualização do perfil.

Mas, primeiro vamos lá na nossa migration para criarmos nossa tabela.

Abra o seu arquivo de migration, no meu caso

2019_10_22_193049_create_profiles_table.php, e adicione o conteúdo do método up abaixo:

```
1 $table->bigIncrements('id');
2 $table->unsignedBigInteger('user_id');
3
4 $table->string('avatar')->nullable();
5 $table->text('about')->nullable();
6
7 $table->string('facebook_link');
8 $table->string('instagram_link');
9 $table->string('site_link');
10
11 $table->timestamps();
12
13 $table->foreign('user_id')->references('id')->on('users');
```

Veja na íntegra toda a migration:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateProfilesTable extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12 }
```



```

12     * @return void
13     */
14     public function up()
15     {
16         Schema::create('profiles', function (Blueprint $table) {
17
18             $table->bigIncrements('id');
19             $table->unsignedBigInteger('user_id');
20
21             $table->string('avatar')->nullable();
22             $table->text('about')->nullable();
23
24             $table->string('facebook_link')->nullable();
25             $table->string('instagram_link')->nullable();
26             $table->string('site_link')->nullable();
27
28             $table->timestamps();
29
30             $table->foreign('user_id')->references('id')->on('users');
31         });
32     }
33
34     /**
35      * Reverse the migrations.
36      *
37      * @return void
38      */
39     public function down()
40     {
41         Schema::dropIfExists('profiles');
42     }
43 }

```

Após isso, basta executar a migration com o comando já conhecido:

```
1 php artisan migrate
```

```

blog(master) X: php artisan migrate
Migrating: 2019_10_22_193049_create_profiles_table
Migrated: 2019_10_22_193049_create_profiles_table (0.1 seconds)
blog(master) X:

```

Perceba que a tabela `profiles` recebe a referência do usuário ao qual pertencerá. Coloquei praticamente todos os campos como `nullable`, para que o usuário possa ter o perfil sem necessitar no primeiro momento de ter as informações completas nele. Obrigatoriamente mesmo, somente a referência `user_id`.

Nesta tabela também teremos a foto do usuário, além do sobre e alguns links de redes sociais. Sobre as fotos, no próximo capítulo iremos abordar a parte de upload e já incrementaremos esse profile bem como a parte de capa das postagens.

Agora vamos partir para as relações.

Criando Relação 1:1 nos Models

O método para mapeamento da relação 1 para 1 é bem simples, o dono da relação, neste caso `User` terá a definição chamando o método `hasOne` (Tem ou Possui Um) já o inverso, `Profile`, terá o método, que já vimos, o `belongsTo` (Pertence a).

Lá no model `User` adicione o seguinte método:

```

1 public function profile()
2 {
3     return $this->hasOne(Profile::class);

```



```
4 }
```

e lá no model profile, adicione:

```
1 public function user()
2 {
3     return $this->belongsTo(User::class);
4 }
```

Agora precisamos criar os métodos do nosso controller e depois adicionar as rotas para acesso. Vamos ao controller primeiramente.

Nosso controller terá apenas dois métodos, um método index e outro update. O método index exibirá um form com os dados do usuário logado e seu perfil e o update será para atualizarmos o perfil deste usuário.

O método index é mais simples então vamos primeiro a ele:

```
1 public function index()
2 {
3     $user = auth()->user();
4
5     if(!$user->profile()->count()) {
6         $user->profile()->create();
7     }
8
9     return view('profile.index', compact('user'));
10 }
```

Recupero o usuário da sessão, como já conhecemos, verifico se ele possui um profile chamando o método da ligação e depois chamando o método count. Este método vai retornar o valor 0 caso o usuário não tenha um perfil criado, ao negar este zero o PHP irá comparar e retornará true, fazendo com que entremos na condição onde

simplesmente criamos um perfil sem nenhum dado, apenas a referência do usuário será adicionada pelo model e chamar o create, já basta para termos um perfil para este usuário.

Logo abaixo, chamo nossa view index.blade.php mandando este usuário logado na sessão para ela. Veja a view:

PS.: Crie a view index.blade.php dentro da pastas views na pasta profile que também precisa ser criada.

```
1 @extends('layouts.app')
2
3 @section('content')
4     <form action="
5         {{route('profile.update')}}" method="post">
6         @csrf
7
8         <div class="form-group">
9             <label>Nome</label>
10
11             <input type="text" name="user[name]" class="form-
12                 control" value="{{ $user\
13                 ->name }}">
14             </div>
15
16             <div class="form-group">
17                 <label>E-mail</label>
18
19                 <input type="text" name="user[email]" class="form-
20                     control" value="{{ $use\
21                     r->email }}">
22             </div>
23
24             <div class="form-group">
```



```

21         <label>Senha</label>
22         <input type="password" name="user[password]" class="form-
23         control" placeholder="Se deseja atualizar sua senha digite aqui a senha
24         nova...">
25     </div>
26     <div class="form-group">
27         <label>Sobre</label>
28         <textarea name="profile[about]" id="" cols="30" rows="10" class="form-
29         control">{{ $user->profile->about }}</textarea>
30     </div>
31
32     <div class="form-group">
33         <label>Facebook</label>
34         <input type="url" name="profile[facebook_link]" class="form-
35         control" value="{{ $user->profile->facebook_link }}">
36     </div>
37     <div class="form-group">
38         <label>Instagram</label>
39         <input type="url" name="profile[instagram_link]" class="form-
40         control" value="{{ $user->profile->instagram_link }}">
41     </div>
42     <div class="form-group">
43         <label>Site</label>

```

```

45         <input type="url" name="profile[site_link]" class="form-
46         control" value="{{ $user->profile->site_link }}">
47     </div>
48
49     <div class="form-group">
50         <button class="btn btn-lg btn-success">Atualizar Meu Perfil</button>
51     </div>
52 </form>
53 @endsection

```

Temos três campos para os dados do usuário: **name**, **email** e **password**. E para o perfil chamo os campos: **about**, **facebook_link**, **instagram_link** e **site_link**.

Perceba que usei uma notação de array nos nomes dos inputs, colocando os campos de user, da seguinte maneira:

- user[name]
- user[email]
- user[password]

E de profile:

- profile[about]
- profile[facebook_link]
- profile[instagram_link]
- profile[site_link]

Quando eu recuperar o campo user e o campo profile lá na request, já terei um array com os campos informados e seus valores de forma simples e direta, no update você vai entender melhor essa necessidade.

Para o campo de senha, não exibir a senha para só atualizarmos a mesma caso o usuário preencha algum valor no campo, basta lê o atributo placeholder do campo de senha do formulário.

Nosso formulário vai apontar para a rota de apelido `profile.update` que ainda não criamos, vamos criar já já mas antes vamos para o método `update` lá no `ProfileController`, vamos lá.

Veja abaixo o método `update` e logo após realizo os comentários:

```
1 public function update(Request $request)
2 {
3     $userData = $request->get('user');
4     $profileData = $request->get('profile');
5
6     try{
7
8         if($userData['password']) {
9             $userData['password'] =
bcrypt($userData['password']);
10         } else {
11             unset($userData['password']);
12         }
13
14         $user = auth()->user();
15
16         $user->update($userData);
17
18         $user->profile()->update($profileData);
19
20         flash('Perfil atualizado com sucesso!')->success();
21         return redirect()->route('profile.index');
22     } catch(\Exception $e) {
23
24     }
```

```
25         $message = 'Erro ao remover categoria!';
26
27         if(env('APP_DEBUG')) {
28             $message = $e->getMessage();
29         }
30
31         flash($message)->warning();
32         return redirect()->back();
33
34     }
35 }
```

Vamos comentar ponto a ponto.

Primeiramente perceba como recupero os arrays referentes aos dados do usuário e referentes ao seu perfil(profile) de forma simples e sem muito esforço por conta das notações que usei no atributo name dos input lá do form. Veja:

```
1 $userData = $request->get('user');
2 $profileData = $request->get('profile');
```

Agora entramos no bloco `try` onde verifico se o campo `password` possui algum valor, se sim, nós alteramos a senha e já encriptamos usando um método helper do Laravel chamado `bcrypt` reescrevendo o valor da chave `password` dentro do array em `$userData`, bem simples não!?

Se a senha não possui valor, ou seja, o usuário não quis mudar a senha eu simplesmente removo ela do array usando o método `unset` do PHP. Veja o trecho:

```
1 if($userData['password']) {
2     $userData['password'] = bcrypt($userData['password']);
3 } else {
4     unset($userData['password']);
```


5 }

Logo após isso, já recupero o usuário logado, e já chamo em seguida o método update que atualizará os dados deste usuário:

```
1 $user = auth()->user();
2
3 $user->update($userData);
```

Em seguida acesso a ligação profile() e uso o método update, que atualizará o perfil do usuário, com os dados do \$profileData. Veja:

```
1 $user->profile()->update($profileData);
```

O restante já conhecemos e vimos, sobre as mensagens e os redirecionamentos, do sucesso mando para a rota de apelido: profile.index (que ainda não existe mas será nosso próximo passo, expor estas rotas). Se erro fazemos o mesmo controle retornando pro momento anterior com a mensagem do erro real se em desenvolvimento ou a mensagem genérica se em produção.

Agora para testarmos, precisamos acessar nosso arquivo web.php lá na pasta routes e adicionar as rotas abaixo:

```
1
Route::prefix('profile')->name('profile.')->group(function(){
2
3
4   Route::get('/', 'ProfileController@index')->name('index');
5
6   Route::post('/', 'ProfileController@update')->name('update')
7
8   ;
9
10  });
11
12  });
13
14  });
15
16  });
```

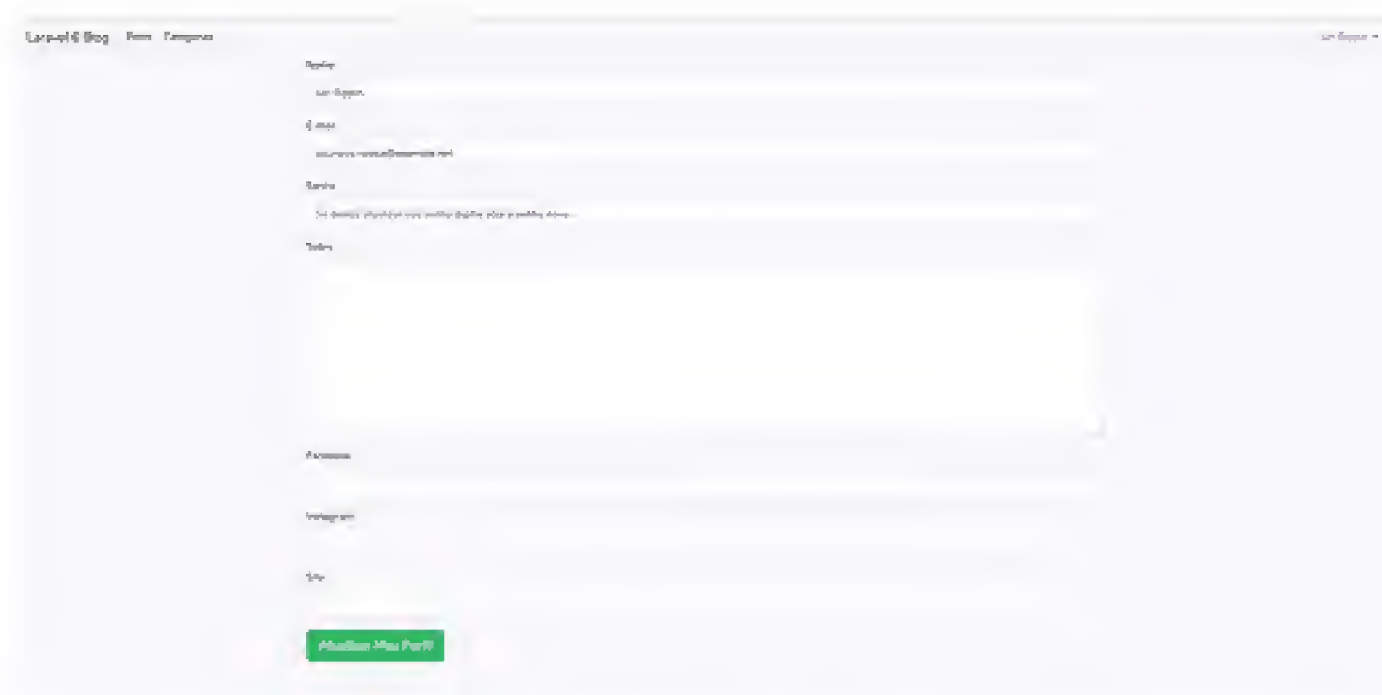
As rotas recebem o prefixo profile e o apelido base profile., neste

grupo temos duas rotas uma get, e outra post. Apontando para o método index e updaterepectivamente.

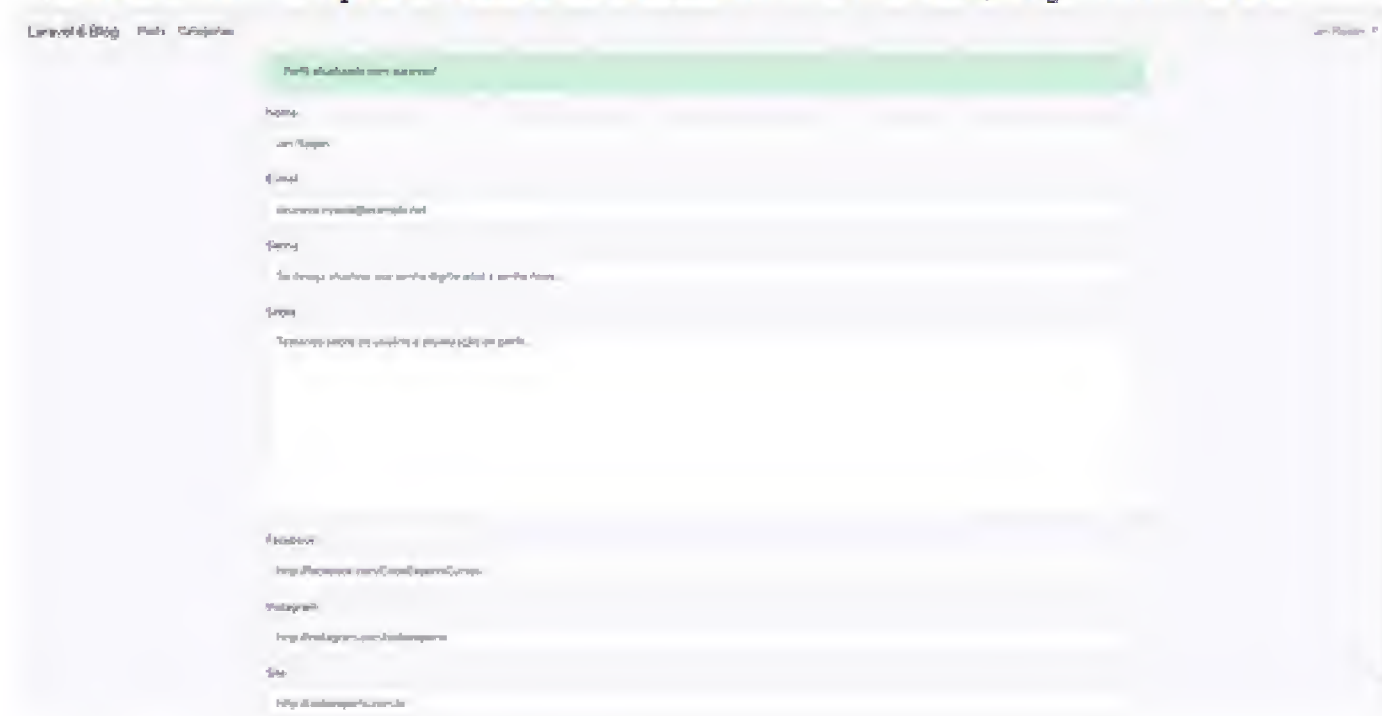
Veja as rotas do admin na íntegra agora:

```
1 Route::group(['middleware' => ['auth']], function(){
2
3
4   Route::prefix('admin')->namespace('Admin')->group(function()
5   {
6
7       Route::resource('posts', 'PostController');
8
9       Route::resource('categories', 'CategoryController');
10
11      Route::prefix('profile')->name('profile.')->group(function()
12      {
13
14          Route::get('/', 'ProfileController@index')->name('index');
15
16          Route::post('/', 'ProfileController@update')->name('update')
17
18          ;
19
20          });
21
22      });
23
24      });
25
26      });
```

Agora podemos acessar em nosso browser o link **http://127.0.0.1:8000/admin/profile**, chegando no formulário abaixo:



Preencha os campos e atualize os dados e submeta, veja o resltado:



Perfil funcionando com sucesso!

Detalhes e melhorias

Bom, fiz algumas adições no nosso layout (app.blade.php) com respeito a parte onde exibimos o nome do usuário logado no menu. Substitua:

```
1 @auth
2 <div class="float-right">
3     <strong>{{auth()->user()->name}}</strong>
4 </div>
5 @endauth
```

Por:

```
1 @auth
2     <ul class="navbar-nav ml-auto">
3         <li class="nav-item dropdown">
4             <a id="navbarDropdown" class="nav-link
5 dropdown-toggle" href="#" rol\
6 e="button" data-toggle="dropdown" aria-
7 haspopup="true" aria-expanded="false" v-pre>
8         {{auth()->user()->name}} <span class="caret"></span>
9             </a>
10         <div class="dropdown-menu dropdown-menu-
11 right" aria-labelledby="navb\
12 arDropdown">
13             <a class="dropdown-item" href="{{
14 route('logout') }}"
15                 onclick="event.preventDefault();
16                 document.getElementById('logout-form').submi\
17 t();">
18                 Sair
19             </a>
```



```

18         <form id="logout-form" action="{{
route('logout') }}" method="POST" style="display: none;"
19         @csrf
20     </form>
21
22     <a class="dropdown-item" href="{{
route('profile.index') }}">
23         Profile
24     </a>
25 </div>
26 </li>
27 </ul>
28 @endauth
29

```

Fiz também as chamadas para os assets buildados no começo deste módulo. Onde temos a chamada para o cdn do bootstrap assim:

```

1 <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/\
2 css/bootstrap.min.css">

```

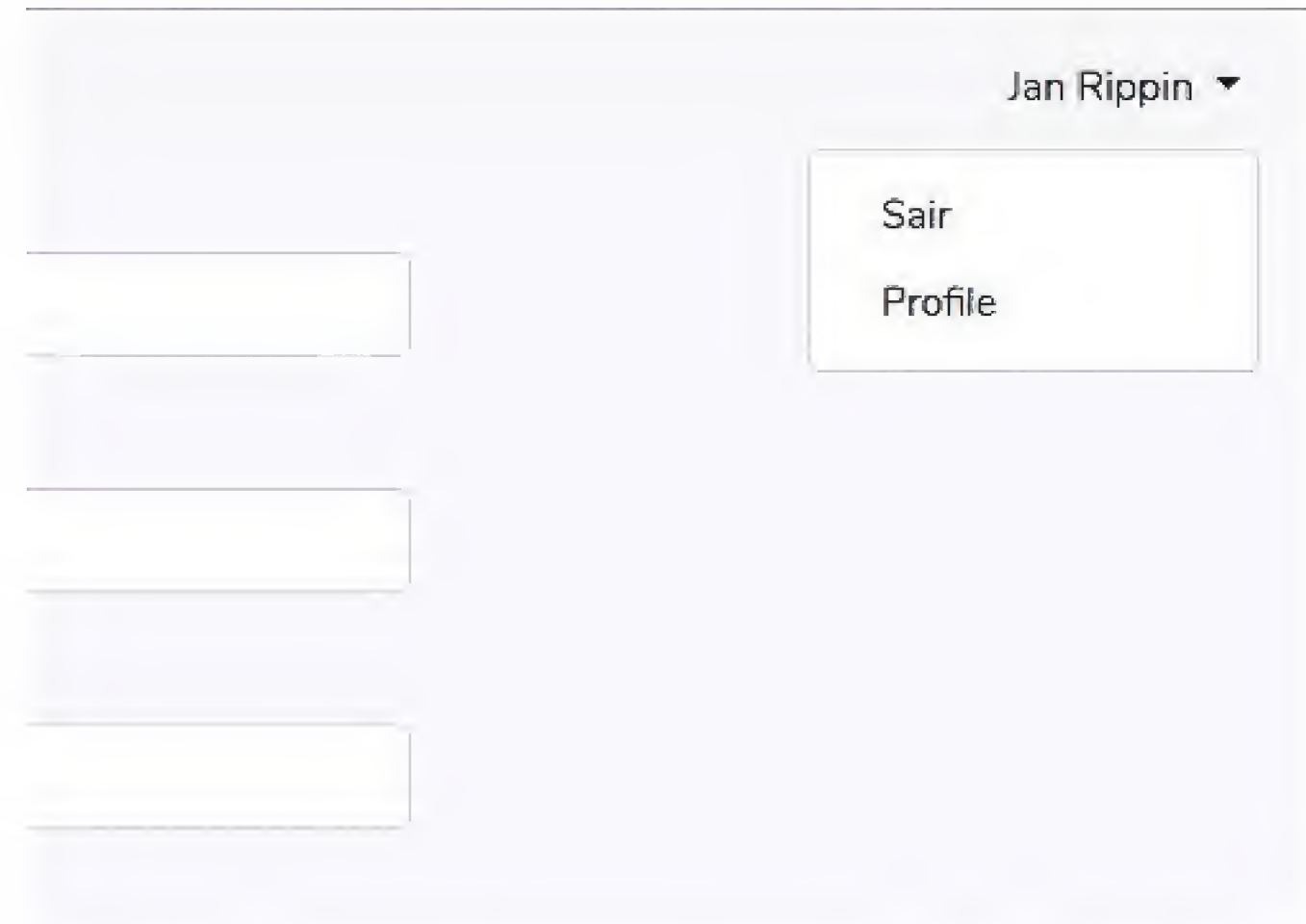
Coloque desta maneira:

```
1 <link rel="stylesheet" href="{{asset('css/app.css')}}">
```

E adicione a chamada para o javascript buildado também antes do fechamento da tag body:

```
1 <script src="{{asset('js/app.js')}}"></script>
```

Para que o dropdown, onde temos os links de sair e de profile, funcionem corretamente. O resultado da mudança do nome do usuário para o dropdown é:



PS.: O link de sair possui um detalhe importante a ser comentado, vamos falar sobre ele então. Veja o trecho dele:

```

1 <a class="dropdown-item" href="{{ route('logout') }}"
2     onclick="event.preventDefault();
3     document.getElementById('logout-
form').submit();">
4     Sair
5 </a>
6
7 <form id="logout-
form" action="{{ route('logout') }}" method="POST" style="display: \

```



```

8 none;">
9     @csrf
10 </form>

```

Como o link de logout só funciona com POST precisamos enviar por meio do elemento `a` esta requisição, por isso que temos a propriedade `onclick` com o conteúdo:

```

1 onclick="event.preventDefault();
2     document.getElementById('logout-form').submit();"

```

Primeiramente ele previne o comportamento padrão do link e depois procura pelo elemento de id `logout-form` e submete ele com o método `submit()`. O elemento que possui o id procurado é justamente um formulário, que está logo abaixo do link.

Esse form que envia a requisição post para logout e encerra nossa sessão. Veja o form:

```

1 <form id="logout-
form" action="{{ route('logout') }}" method="POST" style="display: \
2 none;">
3     @csrf
4 </form>

```

O form está com `display none`, por isso não aparece mas é executado quando clicamos no link Sair.

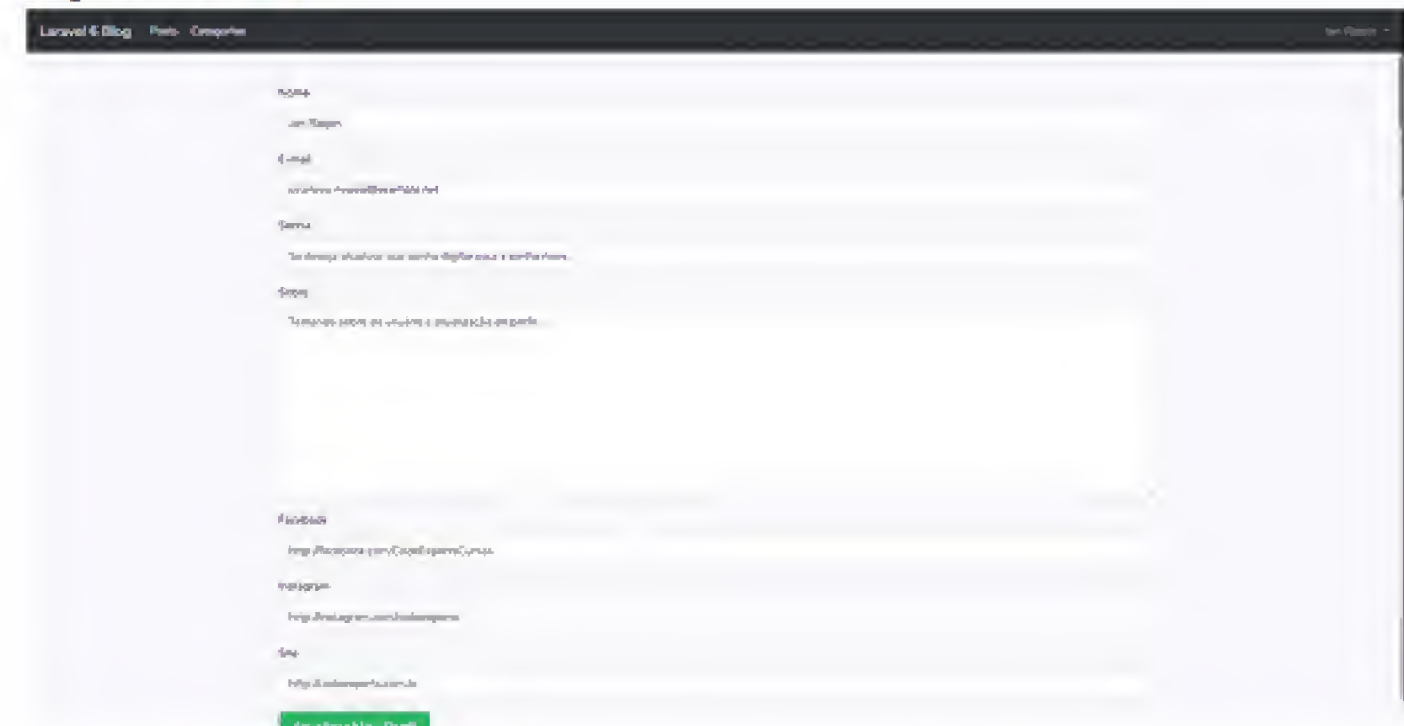
Podemos melhorar ainda o visual do nosso navbar antes de concluirmos nosso capítulo. Procure o elemento `nav` e altere as classes `bg-light` e `navbar-light` por `navbar-dark` e `bg-dark`. E por último, adicione o trecho abaixo, logo após a chamada do css no head, para darmos um espaço do navbar para o conteúdo das páginas:

```

1 <style>
2     .navbar {
3         margin-bottom: 40px;
4     }
5 </style>

```

Veja o resultado:



Segue nosso `app.blade.php` completo e alterado até o momento, veja:

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no,
7         initial-scale=1.0, maximum-\
8         scale=1.0, minimum-scale=1.0">
9     <meta http-equiv="X-UA-Compatible" content="ie=edge">

```



```

9     <title>Gerenciador de Posts</title>
10    <link rel="stylesheet" href="{{asset('css/app.css')}}">
11    <style>
12        .navbar {
13            margin-bottom: 40px;
14        }
15    </style>
16 </head>
17 <body>
18     <nav class="navbar navbar-expand-lg navbar-dark bg-
dark">
19         <a class="navbar-brand" href="/">Laravel 6 Blog</a>
20         <button class="navbar-toggler" type="button" data-
toggle="collapse" data-tar\
21 get="#navbarNavDropdown" aria-
controls="navbarNavDropdown" aria-expanded="false" ari\
22 a-label="Toggle navigation">
23             <span class="navbar-toggler-icon"></span>
24         </button>
25         <div class="collapse navbar-
collapse" id="navbarNavDropdown">
26             @auth
27                 <ul class="navbar-nav">
28                     <li class="nav-item active">
29                         <a class="nav-link" href="{{
route('posts.index') }}">Posts<\
30 /a>
31                     </li>
32                     <li class="nav-item active">
33                         <a class="nav-link" href="{{
route('categories.index') }}">C\
34 ategorias</a>

```

```

35                 </li>
36             </ul>
37         @endauth
38     </div>
39
40
41     @auth
42         <ul class="navbar-nav ml-auto">
43             <li class="nav-item dropdown">
44                 <a id="navbarDropdown" class="nav-
link dropdown-toggle" href\
45 ="#" role="button" data-toggle="dropdown" aria-
haspopup="true" aria-expanded="false"\
46 v-pre>
47                 {{auth()->user()->name}} <span class="caret"></span>
48                 </a>
49
50                 <div class="dropdown-menu dropdown-
menu-right" aria-labelledby\
51 by="navbarDropdown">
52                     <a class="dropdown-
item" href="{{ route('logout') }}"
53
54                     onclick="event.preventDefault();
55 document.getElementById('logout-form\
56 ').submit();">
57                         Sair
58                     </a>
59
60                     <form id="logout-
form" action="{{ route('logout') }}" me\

```



```

60 thod="POST" style="display: none;">
61         @csrf
62     </form>
63
64     <a class="dropdown-
item" href="{ route('profile.index')\
65     }}">
66         Profile
67     </a>
68 </div>
69 </li>
70 </ul>
71 @endauth
72 </nav>
73 <div class="container">
74     @include("flash::message")
75     @yield('content')
76 </div>
77
78 <script src="{{asset('js/app.js')}}"></script>
79 </body>
80 </html>

```

Conclusões

Bom, era isso que queria apresentar neste módulo. A simplicidade de criarmos e manipularmos autenticação em nossos projetos e ainda entendermos como trabalhar com o usuário logado onde realizamos a criação do seu perfil bem como suas relações com o model User, na chamada de 1 para 1.

Você deve ter sentido, ou não, falta do campo avatar no perfil do usuário mas não se preocupe que no próximo capítulo irei abordar upload de

arquivos dentro do Laravel e veremos como adicionar uma foto de capa para uma postagem e também um avatar para o perfil do usuário.

Este capítulo foi mais um capítulo puxado e espero que esteja te ajudando nesta jornada, vamos ficando por aqui mas já partindo para nossa próxima jornada, uploads!

Até lá!

Upload de Arquivos

Vamos entender como funciona o upload de arquivos no Laravel, criando o upload da foto do avatar do perfil e uma capa para as postagens. Primeiramente vamos entender um pouco do funcionamento geral e configurações de upload no Laravel.

Conhecendo Upload no Laravel

Primeiramente podemos recuperar um arquivo, vindo de um input do tipo file, usando o método do request chamado de file. Por exemplo, quando enviarmos lá da tela do perfil do usuário o campo avatar:

```
1 $avatar = $request->file('avatar');
```

Poderíamos acessar também, da maneira abaixo:

```
1 $avatar = $request->avatar;
```

Este método irá retornar um objeto do tipo Illuminate\Http\UploadedFile com as informações do arquivo. Onde poderemos recuperar diversas informações, como por exemplo o nome real do arquivo enviado:

```
1 $avatar->getClientOriginalName();
```

Ou mesmo a extensão do arquivo:

```
1 $avatar->extension();
```

Podemos de cara já realizar o upload deste arquivo, sem muito esforço, simplesmente chamando o método store:

```
1 $path = $avatar->store('avatars');
```

O método store por padrão pegará o caminho do drive default configurado lá no filesystems.php dentro da pasta config do projeto. E moverá a imagem para esta pasta padrão e ainda criando a pasta avatars caso não exista e o melhor de tudo, ele já manda o arquivo com um nome modificado, usando um hash para evitar conflitos de mesmo nome para arquivos. O caminho padrão é storage/app mas se você quiser salvar em outro caminho por disco, como é chamado a configuração, você precisa informar o segundo parâmetro.

Por exemplo, vamos salvar as imagens no public do storage onde futuramente linkaremos com a pasta public do projeto. O drive que representa este caminho é chamado de public que aponta para storage/app/public e se quisermos referenciar ele, temos que chamar como abaixo:

```
1 $path = $avatar->store('avatars', 'public');
```

O retorno do método store é o nome da imagem mas a pasta, por exemplo:

```
1 avatars/lowy00Yzb5CDoyMEN1QGQ86jlyMCvJJm0fEDa5Ue.jpeg
```

Esse nome + caminho que salvaremos na coluna do avatar por exemplo. Agora vamos conhecer as configurações de armazenamento lá no filesystems.php.

config/filesystems.php:

```
1 <?php
2
3 return [
4
5     /*
6
7     .....
8     */
9 ]
```



```

-----
7      | Default Filesystem Disk
8
9      |-----
10     |
11     | Here you may specify the default filesystem disk that
12     | should be used
13     | by the framework. The "local" disk, as well as a
14     | variety of cloud
15     | based disks are available to your application. Just
16     | store away!
17     |
18     |*/
19
20     |-----
21     |
22     | Default Cloud Filesystem Disk
23     |
24     | Many applications store files both locally and in the
25     | cloud. For this
26     | reason, you may specify a default "cloud" driver
27     | here. This driver
28     | will be bound as the Cloud disk implementation in the
29     | container.
30     |
31     |

```

```

32     |*/
33     |
34     |-----
35     |
36     | Filesystem Disks
37     |
38     |-----
39     |
40     | Here you may configure as many filesystem "disks" as
41     | you wish, and you
42     | may even configure multiple disks of the same driver.
43     | Defaults have
44     | been setup for each driver as an example of the
45     | required options.
46     |
47     | Supported Drivers: "local", "ftp", "sftp", "s3"
48     |
49     |*/
50
51     'disks' => [
52         'local' => [
53             'driver' => 'local',
54             'root' => storage_path('app'),
55         ],
56         'public' => [
57             'driver' => 'local',
58             'root' => storage_path('app/public'),
59         ],
60     ],
61
62     'cloud' => env('FILESYSTEM_CLOUD', 's3'),
63
64     /*
65     |-----
66     |
67     | Filesystem Disks
68     |
69     |-----
70     |
71     | Here you may configure as many filesystem "disks" as
72     | you wish, and you
73     | may even configure multiple disks of the same driver.
74     | Defaults have
75     | been setup for each driver as an example of the
76     | required options.
77     |
78     | Supported Drivers: "local", "ftp", "sftp", "s3"
79     |
80     |*/
81
82     'disks' => [
83         'local' => [
84             'driver' => 'local',
85             'root' => storage_path('app'),
86         ],
87         'public' => [
88             'driver' => 'local',
89             'root' => storage_path('app/public'),
90         ],
91     ],
92
93     'cloud' => env('FILESYSTEM_CLOUD', 's3'),
94
95     /*
96     |-----
97     |
98     | Filesystem Disks
99     |
100    |-----
101    |
102    | Here you may configure as many filesystem "disks" as
103    | you wish, and you
104    | may even configure multiple disks of the same driver.
105    | Defaults have
106    | been setup for each driver as an example of the
107    | required options.
108    |
109    | Supported Drivers: "local", "ftp", "sftp", "s3"
110    |
111    |*/
112
113    'disks' => [
114        'local' => [
115            'driver' => 'local',
116            'root' => storage_path('app'),
117        ],
118        'public' => [
119            'driver' => 'local',
120            'root' => storage_path('app/public'),
121        ],
122    ],
123
124    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
125
126    /*
127    |-----
128    |
129    | Filesystem Disks
130    |
131    |-----
132    |
133    | Here you may configure as many filesystem "disks" as
134    | you wish, and you
135    | may even configure multiple disks of the same driver.
136    | Defaults have
137    | been setup for each driver as an example of the
138    | required options.
139    |
140    | Supported Drivers: "local", "ftp", "sftp", "s3"
141    |
142    |*/
143
144    'disks' => [
145        'local' => [
146            'driver' => 'local',
147            'root' => storage_path('app'),
148        ],
149        'public' => [
150            'driver' => 'local',
151            'root' => storage_path('app/public'),
152        ],
153    ],
154
155    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
156
157    /*
158    |-----
159    |
160    | Filesystem Disks
161    |
162    |-----
163    |
164    | Here you may configure as many filesystem "disks" as
165    | you wish, and you
166    | may even configure multiple disks of the same driver.
167    | Defaults have
168    | been setup for each driver as an example of the
169    | required options.
170    |
171    | Supported Drivers: "local", "ftp", "sftp", "s3"
172    |
173    |*/
174
175    'disks' => [
176        'local' => [
177            'driver' => 'local',
178            'root' => storage_path('app'),
179        ],
180        'public' => [
181            'driver' => 'local',
182            'root' => storage_path('app/public'),
183        ],
184    ],
185
186    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
187
188    /*
189    |-----
190    |
191    | Filesystem Disks
192    |
193    |-----
194    |
195    | Here you may configure as many filesystem "disks" as
196    | you wish, and you
197    | may even configure multiple disks of the same driver.
198    | Defaults have
199    | been setup for each driver as an example of the
200    | required options.
201    |
202    | Supported Drivers: "local", "ftp", "sftp", "s3"
203    |
204    |*/
205
206    'disks' => [
207        'local' => [
208            'driver' => 'local',
209            'root' => storage_path('app'),
210        ],
211        'public' => [
212            'driver' => 'local',
213            'root' => storage_path('app/public'),
214        ],
215    ],
216
217    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
218
219    /*
220    |-----
221    |
222    | Filesystem Disks
223    |
224    |-----
225    |
226    | Here you may configure as many filesystem "disks" as
227    | you wish, and you
228    | may even configure multiple disks of the same driver.
229    | Defaults have
230    | been setup for each driver as an example of the
231    | required options.
232    |
233    | Supported Drivers: "local", "ftp", "sftp", "s3"
234    |
235    |*/
236
237    'disks' => [
238        'local' => [
239            'driver' => 'local',
240            'root' => storage_path('app'),
241        ],
242        'public' => [
243            'driver' => 'local',
244            'root' => storage_path('app/public'),
245        ],
246    ],
247
248    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
249
250    /*
251    |-----
252    |
253    | Filesystem Disks
254    |
255    |-----
256    |
257    | Here you may configure as many filesystem "disks" as
258    | you wish, and you
259    | may even configure multiple disks of the same driver.
260    | Defaults have
261    | been setup for each driver as an example of the
262    | required options.
263    |
264    | Supported Drivers: "local", "ftp", "sftp", "s3"
265    |
266    |*/
267
268    'disks' => [
269        'local' => [
270            'driver' => 'local',
271            'root' => storage_path('app'),
272        ],
273        'public' => [
274            'driver' => 'local',
275            'root' => storage_path('app/public'),
276        ],
277    ],
278
279    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
280
281    /*
282    |-----
283    |
284    | Filesystem Disks
285    |
286    |-----
287    |
288    | Here you may configure as many filesystem "disks" as
289    | you wish, and you
290    | may even configure multiple disks of the same driver.
291    | Defaults have
292    | been setup for each driver as an example of the
293    | required options.
294    |
295    | Supported Drivers: "local", "ftp", "sftp", "s3"
296    |
297    |*/
298
299    'disks' => [
300        'local' => [
301            'driver' => 'local',
302            'root' => storage_path('app'),
303        ],
304        'public' => [
305            'driver' => 'local',
306            'root' => storage_path('app/public'),
307        ],
308    ],
309
310    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
311
312    /*
313    |-----
314    |
315    | Filesystem Disks
316    |
317    |-----
318    |
319    | Here you may configure as many filesystem "disks" as
320    | you wish, and you
321    | may even configure multiple disks of the same driver.
322    | Defaults have
323    | been setup for each driver as an example of the
324    | required options.
325    |
326    | Supported Drivers: "local", "ftp", "sftp", "s3"
327    |
328    |*/
329
330    'disks' => [
331        'local' => [
332            'driver' => 'local',
333            'root' => storage_path('app'),
334        ],
335        'public' => [
336            'driver' => 'local',
337            'root' => storage_path('app/public'),
338        ],
339    ],
340
341    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
342
343    /*
344    |-----
345    |
346    | Filesystem Disks
347    |
348    |-----
349    |
350    | Here you may configure as many filesystem "disks" as
351    | you wish, and you
352    | may even configure multiple disks of the same driver.
353    | Defaults have
354    | been setup for each driver as an example of the
355    | required options.
356    |
357    | Supported Drivers: "local", "ftp", "sftp", "s3"
358    |
359    |*/
360
361    'disks' => [
362        'local' => [
363            'driver' => 'local',
364            'root' => storage_path('app'),
365        ],
366        'public' => [
367            'driver' => 'local',
368            'root' => storage_path('app/public'),
369        ],
370    ],
371
372    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
373
374    /*
375    |-----
376    |
377    | Filesystem Disks
378    |
379    |-----
380    |
381    | Here you may configure as many filesystem "disks" as
382    | you wish, and you
383    | may even configure multiple disks of the same driver.
384    | Defaults have
385    | been setup for each driver as an example of the
386    | required options.
387    |
388    | Supported Drivers: "local", "ftp", "sftp", "s3"
389    |
390    |*/
391
392    'disks' => [
393        'local' => [
394            'driver' => 'local',
395            'root' => storage_path('app'),
396        ],
397        'public' => [
398            'driver' => 'local',
399            'root' => storage_path('app/public'),
400        ],
401    ],
402
403    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
404
405    /*
406    |-----
407    |
408    | Filesystem Disks
409    |
410    |-----
411    |
412    | Here you may configure as many filesystem "disks" as
413    | you wish, and you
414    | may even configure multiple disks of the same driver.
415    | Defaults have
416    | been setup for each driver as an example of the
417    | required options.
418    |
419    | Supported Drivers: "local", "ftp", "sftp", "s3"
420    |
421    |*/
422
423    'disks' => [
424        'local' => [
425            'driver' => 'local',
426            'root' => storage_path('app'),
427        ],
428        'public' => [
429            'driver' => 'local',
430            'root' => storage_path('app/public'),
431        ],
432    ],
433
434    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
435
436    /*
437    |-----
438    |
439    | Filesystem Disks
440    |
441    |-----
442    |
443    | Here you may configure as many filesystem "disks" as
444    | you wish, and you
445    | may even configure multiple disks of the same driver.
446    | Defaults have
447    | been setup for each driver as an example of the
448    | required options.
449    |
450    | Supported Drivers: "local", "ftp", "sftp", "s3"
451    |
452    |*/
453
454    'disks' => [
455        'local' => [
456            'driver' => 'local',
457            'root' => storage_path('app'),
458        ],
459        'public' => [
460            'driver' => 'local',
461            'root' => storage_path('app/public'),
462        ],
463    ],
464
465    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
466
467    /*
468    |-----
469    |
470    | Filesystem Disks
471    |
472    |-----
473    |
474    | Here you may configure as many filesystem "disks" as
475    | you wish, and you
476    | may even configure multiple disks of the same driver.
477    | Defaults have
478    | been setup for each driver as an example of the
479    | required options.
480    |
481    | Supported Drivers: "local", "ftp", "sftp", "s3"
482    |
483    |*/
484
485    'disks' => [
486        'local' => [
487            'driver' => 'local',
488            'root' => storage_path('app'),
489        ],
490        'public' => [
491            'driver' => 'local',
492            'root' => storage_path('app/public'),
493        ],
494    ],
495
496    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
497
498    /*
499    |-----
500    |
501    | Filesystem Disks
502    |
503    |-----
504    |
505    | Here you may configure as many filesystem "disks" as
506    | you wish, and you
507    | may even configure multiple disks of the same driver.
508    | Defaults have
509    | been setup for each driver as an example of the
510    | required options.
511    |
512    | Supported Drivers: "local", "ftp", "sftp", "s3"
513    |
514    |*/
515
516    'disks' => [
517        'local' => [
518            'driver' => 'local',
519            'root' => storage_path('app'),
520        ],
521        'public' => [
522            'driver' => 'local',
523            'root' => storage_path('app/public'),
524        ],
525    ],
526
527    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
528
529    /*
530    |-----
531    |
532    | Filesystem Disks
533    |
534    |-----
535    |
536    | Here you may configure as many filesystem "disks" as
537    | you wish, and you
538    | may even configure multiple disks of the same driver.
539    | Defaults have
540    | been setup for each driver as an example of the
541    | required options.
542    |
543    | Supported Drivers: "local", "ftp", "sftp", "s3"
544    |
545    |*/
546
547    'disks' => [
548        'local' => [
549            'driver' => 'local',
550            'root' => storage_path('app'),
551        ],
552        'public' => [
553            'driver' => 'local',
554            'root' => storage_path('app/public'),
555        ],
556    ],
557
558    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
559
560    /*
561    |-----
562    |
563    | Filesystem Disks
564    |
565    |-----
566    |
567    | Here you may configure as many filesystem "disks" as
568    | you wish, and you
569    | may even configure multiple disks of the same driver.
570    | Defaults have
571    | been setup for each driver as an example of the
572    | required options.
573    |
574    | Supported Drivers: "local", "ftp", "sftp", "s3"
575    |
576    |*/
577
578    'disks' => [
579        'local' => [
580            'driver' => 'local',
581            'root' => storage_path('app'),
582        ],
583        'public' => [
584            'driver' => 'local',
585            'root' => storage_path('app/public'),
586        ],
587    ],
588
589    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
590
591    /*
592    |-----
593    |
594    | Filesystem Disks
595    |
596    |-----
597    |
598    | Here you may configure as many filesystem "disks" as
599    | you wish, and you
600    | may even configure multiple disks of the same driver.
601    | Defaults have
602    | been setup for each driver as an example of the
603    | required options.
604    |
605    | Supported Drivers: "local", "ftp", "sftp", "s3"
606    |
607    |*/
608
609    'disks' => [
610        'local' => [
611            'driver' => 'local',
612            'root' => storage_path('app'),
613        ],
614        'public' => [
615            'driver' => 'local',
616            'root' => storage_path('app/public'),
617        ],
618    ],
619
620    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
621
622    /*
623    |-----
624    |
625    | Filesystem Disks
626    |
627    |-----
628    |
629    | Here you may configure as many filesystem "disks" as
630    | you wish, and you
631    | may even configure multiple disks of the same driver.
632    | Defaults have
633    | been setup for each driver as an example of the
634    | required options.
635    |
636    | Supported Drivers: "local", "ftp", "sftp", "s3"
637    |
638    |*/
639
640    'disks' => [
641        'local' => [
642            'driver' => 'local',
643            'root' => storage_path('app'),
644        ],
645        'public' => [
646            'driver' => 'local',
647            'root' => storage_path('app/public'),
648        ],
649    ],
650
651    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
652
653    /*
654    |-----
655    |
656    | Filesystem Disks
657    |
658    |-----
659    |
660    | Here you may configure as many filesystem "disks" as
661    | you wish, and you
662    | may even configure multiple disks of the same driver.
663    | Defaults have
664    | been setup for each driver as an example of the
665    | required options.
666    |
667    | Supported Drivers: "local", "ftp", "sftp", "s3"
668    |
669    |*/
670
671    'disks' => [
672        'local' => [
673            'driver' => 'local',
674            'root' => storage_path('app'),
675        ],
676        'public' => [
677            'driver' => 'local',
678            'root' => storage_path('app/public'),
679        ],
680    ],
681
682    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
683
684    /*
685    |-----
686    |
687    | Filesystem Disks
688    |
689    |-----
690    |
691    | Here you may configure as many filesystem "disks" as
692    | you wish, and you
693    | may even configure multiple disks of the same driver.
694    | Defaults have
695    | been setup for each driver as an example of the
696    | required options.
697    |
698    | Supported Drivers: "local", "ftp", "sftp", "s3"
699    |
700    |*/
701
702    'disks' => [
703        'local' => [
704            'driver' => 'local',
705            'root' => storage_path('app'),
706        ],
707        'public' => [
708            'driver' => 'local',
709            'root' => storage_path('app/public'),
710        ],
711    ],
712
713    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
714
715    /*
716    |-----
717    |
718    | Filesystem Disks
719    |
720    |-----
721    |
722    | Here you may configure as many filesystem "disks" as
723    | you wish, and you
724    | may even configure multiple disks of the same driver.
725    | Defaults have
726    | been setup for each driver as an example of the
727    | required options.
728    |
729    | Supported Drivers: "local", "ftp", "sftp", "s3"
730    |
731    |*/
732
733    'disks' => [
734        'local' => [
735            'driver' => 'local',
736            'root' => storage_path('app'),
737        ],
738        'public' => [
739            'driver' => 'local',
740            'root' => storage_path('app/public'),
741        ],
742    ],
743
744    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
745
746    /*
747    |-----
748    |
749    | Filesystem Disks
750    |
751    |-----
752    |
753    | Here you may configure as many filesystem "disks" as
754    | you wish, and you
755    | may even configure multiple disks of the same driver.
756    | Defaults have
757    | been setup for each driver as an example of the
758    | required options.
759    |
760    | Supported Drivers: "local", "ftp", "sftp", "s3"
761    |
762    |*/
763
764    'disks' => [
765        'local' => [
766            'driver' => 'local',
767            'root' => storage_path('app'),
768        ],
769        'public' => [
770            'driver' => 'local',
771            'root' => storage_path('app/public'),
772        ],
773    ],
774
775    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
776
777    /*
778    |-----
779    |
780    | Filesystem Disks
781    |
782    |-----
783    |
784    | Here you may configure as many filesystem "disks" as
785    | you wish, and you
786    | may even configure multiple disks of the same driver.
787    | Defaults have
788    | been setup for each driver as an example of the
789    | required options.
790    |
791    | Supported Drivers: "local", "ftp", "sftp", "s3"
792    |
793    |*/
794
795    'disks' => [
796        'local' => [
797            'driver' => 'local',
798            'root' => storage_path('app'),
799        ],
800        'public' => [
801            'driver' => 'local',
802            'root' => storage_path('app/public'),
803        ],
804    ],
805
806    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
807
808    /*
809    |-----
810    |
811    | Filesystem Disks
812    |
813    |-----
814    |
815    | Here you may configure as many filesystem "disks" as
816    | you wish, and you
817    | may even configure multiple disks of the same driver.
818    | Defaults have
819    | been setup for each driver as an example of the
820    | required options.
821    |
822    | Supported Drivers: "local", "ftp", "sftp", "s3"
823    |
824    |*/
825
826    'disks' => [
827        'local' => [
828            'driver' => 'local',
829            'root' => storage_path('app'),
830        ],
831        'public' => [
832            'driver' => 'local',
833            'root' => storage_path('app/public'),
834        ],
835    ],
836
837    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
838
839    /*
840    |-----
841    |
842    | Filesystem Disks
843    |
844    |-----
845    |
846    | Here you may configure as many filesystem "disks" as
847    | you wish, and you
848    | may even configure multiple disks of the same driver.
849    | Defaults have
850    | been setup for each driver as an example of the
851    | required options.
852    |
853    | Supported Drivers: "local", "ftp", "sftp", "s3"
854    |
855    |*/
856
857    'disks' => [
858        'local' => [
859            'driver' => 'local',
860            'root' => storage_path('app'),
861        ],
862        'public' => [
863            'driver' => 'local',
864            'root' => storage_path('app/public'),
865        ],
866    ],
867
868    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
869
870    /*
871    |-----
872    |
873    | Filesystem Disks
874    |
875    |-----
876    |
877    | Here you may configure as many filesystem "disks" as
878    | you wish, and you
879    | may even configure multiple disks of the same driver.
880    | Defaults have
881    | been setup for each driver as an example of the
882    | required options.
883    |
884    | Supported Drivers: "local", "ftp", "sftp", "s3"
885    |
886    |*/
887
888    'disks' => [
889        'local' => [
890            'driver' => 'local',
891            'root' => storage_path('app'),
892        ],
893        'public' => [
894            'driver' => 'local',
895            'root' => storage_path('app/public'),
896        ],
897    ],
898
899    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
900
901    /*
902    |-----
903    |
904    | Filesystem Disks
905    |
906    |-----
907    |
908    | Here you may configure as many filesystem "disks" as
909    | you wish, and you
910    | may even configure multiple disks of the same driver.
911    | Defaults have
912    | been setup for each driver as an example of the
913    | required options.
914    |
915    | Supported Drivers: "local", "ftp", "sftp", "s3"
916    |
917    |*/
918
919    'disks' => [
920        'local' => [
921            'driver' => 'local',
922            'root' => storage_path('app'),
923        ],
924        'public' => [
925            'driver' => 'local',
926            'root' => storage_path('app/public'),
927        ],
928    ],
929
930    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
931
932    /*
933    |-----
934    |
935    | Filesystem Disks
936    |
937    |-----
938    |
939    | Here you may configure as many filesystem "disks" as
940    | you wish, and you
941    | may even configure multiple disks of the same driver.
942    | Defaults have
943    | been setup for each driver as an example of the
944    | required options.
945    |
946    | Supported Drivers: "local", "ftp", "sftp", "s3"
947    |
948    |*/
949
950    'disks' => [
951        'local' => [
952            'driver' => 'local',
953            'root' => storage_path('app'),
954        ],
955        'public' => [
956            'driver' => 'local',
957            'root' => storage_path('app/public'),
958        ],
959    ],
960
961    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
962
963    /*
964    |-----
965    |
966    | Filesystem Disks
967    |
968    |-----
969    |
970    | Here you may configure as many filesystem "disks" as
971    | you wish, and you
972    | may even configure multiple disks of the same driver.
973    | Defaults have
974    | been setup for each driver as an example of the
975    | required options.
976    |
977    | Supported Drivers: "local", "ftp", "sftp", "s3"
978    |
979    |*/
980
981    'disks' => [
982        'local' => [
983            'driver' => 'local',
984            'root' => storage_path('app'),
985        ],
986        'public' => [
987            'driver' => 'local',
988            'root' => storage_path('app/public'),
989        ],
990    ],
991
992    'cloud' => env('FILESYSTEM_CLOUD', 's3'),
993
994    /*
995    |-----
996    |
997    | Filesystem Disks
998    |
999    |-----
1000   |
1001   | Here you may configure as many filesystem "disks" as
1002   | you wish, and you
1003   | may even configure multiple disks of the same driver.
1004   | Defaults have
1005   | been setup for each driver as an example of the
1006   | required options.
1007   |
1008   | Supported Drivers: "local", "ftp", "sftp", "s3"
1009   |
1010   |*/
1011
1012   'disks' => [
1013       'local' => [
1014           'driver' => 'local',
1015           'root' => storage_path('app'),
1016       ],
1017       'public' => [
1018           'driver' => 'local',
1019           'root' => storage_path('app/public'),
1020       ],
1021   ],
1022
1023   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1024
1025   /*
1026   |-----
1027   |
1028   | Filesystem Disks
1029   |
1030   |-----
1031   |
1032   | Here you may configure as many filesystem "disks" as
1033   | you wish, and you
1034   | may even configure multiple disks of the same driver.
1035   | Defaults have
1036   | been setup for each driver as an example of the
1037   | required options.
1038   |
1039   | Supported Drivers: "local", "ftp", "sftp", "s3"
1040   |
1041   |*/
1042
1043   'disks' => [
1044       'local' => [
1045           'driver' => 'local',
1046           'root' => storage_path('app'),
1047       ],
1048       'public' => [
1049           'driver' => 'local',
1050           'root' => storage_path('app/public'),
1051       ],
1052   ],
1053
1054   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1055
1056   /*
1057   |-----
1058   |
1059   | Filesystem Disks
1060   |
1061   |-----
1062   |
1063   | Here you may configure as many filesystem "disks" as
1064   | you wish, and you
1065   | may even configure multiple disks of the same driver.
1066   | Defaults have
1067   | been setup for each driver as an example of the
1068   | required options.
1069   |
1070   | Supported Drivers: "local", "ftp", "sftp", "s3"
1071   |
1072   |*/
1073
1074   'disks' => [
1075       'local' => [
1076           'driver' => 'local',
1077           'root' => storage_path('app'),
1078       ],
1079       'public' => [
1080           'driver' => 'local',
1081           'root' => storage_path('app/public'),
1082       ],
1083   ],
1084
1085   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1086
1087   /*
1088   |-----
1089   |
1090   | Filesystem Disks
1091   |
1092   |-----
1093   |
1094   | Here you may configure as many filesystem "disks" as
1095   | you wish, and you
1096   | may even configure multiple disks of the same driver.
1097   | Defaults have
1098   | been setup for each driver as an example of the
1099   | required options.
1100   |
1101   | Supported Drivers: "local", "ftp", "sftp", "s3"
1102   |
1103   |*/
1104
1105   'disks' => [
1106       'local' => [
1107           'driver' => 'local',
1108           'root' => storage_path('app'),
1109       ],
1110       'public' => [
1111           'driver' => 'local',
1112           'root' => storage_path('app/public'),
1113       ],
1114   ],
1115
1116   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1117
1118   /*
1119   |-----
1120   |
1121   | Filesystem Disks
1122   |
1123   |-----
1124   |
1125   | Here you may configure as many filesystem "disks" as
1126   | you wish, and you
1127   | may even configure multiple disks of the same driver.
1128   | Defaults have
1129   | been setup for each driver as an example of the
1130   | required options.
1131   |
1132   | Supported Drivers: "local", "ftp", "sftp", "s3"
1133   |
1134   |*/
1135
1136   'disks' => [
1137       'local' => [
1138           'driver' => 'local',
1139           'root' => storage_path('app'),
1140       ],
1141       'public' => [
1142           'driver' => 'local',
1143           'root' => storage_path('app/public'),
1144       ],
1145   ],
1146
1147   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1148
1149   /*
1150   |-----
1151   |
1152   | Filesystem Disks
1153   |
1154   |-----
1155   |
1156   | Here you may configure as many filesystem "disks" as
1157   | you wish, and you
1158   | may even configure multiple disks of the same driver.
1159   | Defaults have
1160   | been setup for each driver as an example of the
1161   | required options.
1162   |
1163   | Supported Drivers: "local", "ftp", "sftp", "s3"
1164   |
1165   |*/
1166
1167   'disks' => [
1168       'local' => [
1169           'driver' => 'local',
1170           'root' => storage_path('app'),
1171       ],
1172       'public' => [
1173           'driver' => 'local',
1174           'root' => storage_path('app/public'),
1175       ],
1176   ],
1177
1178   'cloud' => env('FILESYSTEM_CLOUD', 's3'),
1179
1180   /*
1181   |-----
1182   |
1183   | Filesystem Disks
1184   |
1185   |-----
1186   |
1187   | Here you may configure as many filesystem "disks" as
1188   | you wish, and you
1189   | may even configure multiple disks of the same driver.
1190   | Defaults have
1191   | been setup for each driver as an example of the
1192   | required options.
1193   |
1194   | Supported Drivers: "local", "ftp", "sftp", "s3
```



```

54         'url' => env('APP_URL').'/storage',
55         'visibility' => 'public',
56     ],
57
58     's3' => [
59         'driver' => 's3',
60         'key' => env('AWS_ACCESS_KEY_ID'),
61         'secret' => env('AWS_SECRET_ACCESS_KEY'),
62         'region' => env('AWS_DEFAULT_REGION'),
63         'bucket' => env('AWS_BUCKET'),
64         'url' => env('AWS_URL'),
65     ],
66 ],
67 ],
68
69 ];

```

Perceba que temos a chave default logo de cara, que espera o valor vindo lá do .env na variável `FILESYSTEM_DRIVER`, se ela não existir lá, teremos por padrão o valor `local` como disco para salvarmos nossos arquivos.

Veja:

```
1 'default' => env('FILESYSTEM_DRIVER', 'local'),
```

Temos também o mesmo pensamento mas para o arquivo que pode ser salvo na nuvem, neste caso no S3 da Amazon (AWS). Veja:

```
1 'cloud' => env('FILESYSTEM_CLOUD', 's3'),
```

Obs.: O Laravel já vem praticamente pronto para que você possa realizar upload no serviço de storage da Amazon, o S3.

Logo, seguindo pelo arquivo, temos os discos (**disks**) disponíveis e

configurados para armazenamento de arquivos. São eles:

- **local**: pasta app dentro de storage;
- **public**: pasta app/public dentro de storage;
- **s3**: configurações do seu bucket S3 serão necessárias para armazenamento na nuvem.

Veja os discos abaixo:

```

1 'disks' => [
2
3     'local' => [
4         'driver' => 'local',
5         'root' => storage_path('app'),
6     ],
7
8     'public' => [
9         'driver' => 'local',
10        'root' => storage_path('app/public'),
11        'url' => env('APP_URL').'/storage',
12        'visibility' => 'public',
13    ],
14
15    's3' => [
16        'driver' => 's3',
17        'key' => env('AWS_ACCESS_KEY_ID'),
18        'secret' => env('AWS_SECRET_ACCESS_KEY'),
19        'region' => env('AWS_DEFAULT_REGION'),
20        'bucket' => env('AWS_BUCKET'),
21        'url' => env('AWS_URL'),
22    ],
23
24 ],

```

Cada um têm o nome do driver, o caminho, url e a visibilidade,

entretanto, o S3, como é um servidor e serviço remoto e ainda privado necessita de configurações extras para que você possa realizar o upload neste disco.

Conhecendo estes pontos vamos adicionar o upload de arquivos em nosso perfil do usuário e depois em posts.

Upload de Foto Perfil do Usuário

Primeiramente adicione o input do tipo file lá no formulário do perfil em `resources/views/profile/index.blade.php`:

```
1 <div class="form-group">
2     <label>Avatar</label>
3     <input type="file" name="avatar">
4 </div>
```

PS.: coloquei logo após o campo sobre.

Adicionado o campo, precisamos permitir que nossa requisição envie esta imagem para nosso controller adicionando o `enctype` `multipart/form-data` que diz para nossa requisição não mexer ou codificar os dados do nosso form mantendo assim nossa imagem.

Adicione o trecho na tag de abertura do formulário:

```
1 enctype="multipart/form-data"
```

ficando assim:

```
1 <form action="{{route('profile.update')}}" method="post"
  enctype="multipart/form-data">
2 a">
```

Lá no controller `ProfileController`, logo após a recuperação do usuário da sessão no método `update` adicione o trecho abaixo:

```
1 if($request->hasFile('avatar')) {
2
3     $profileData['avatar'] =
  $request->file('avatar')->store('avatars', 'public');
4
5 } else {
6     unset($profileData['avatar']);
7 }
```

Primeiramente verifico se a request possui o arquivo da imagem usando o método `hasFile` informando o nome do input, existindo, eu crio a chave `avatar` no array dentro de `$profileData` e uso os métodos `file` para recuperar a imagem enviada e o método `store` logo em seguida para upload, passando o nome da pasta `avatars` e o disco `public`, que resultará em nossa imagem salva dentro de `storage/app/public/avatars`.

O retorno deste upload eu mando para a chave `avatar` recém criada em `$profileData`, que ao atualizarmos o perfil receberá o nome da imagem bem como o nome da pasta `avatars` junto. Agora se formos ao nosso formulário, podemos testar o envio de uma foto para o perfil do usuário.

Pera, pera!!

Antes de testarmos, precisamos realizar mais uma melhoria no upload `rsrsrs`. Como vamos atualizar a foto do usuário precisamos remover a foto anterior, o arquivo no caso. Para isso precisamos usar o objeto `storage` para interagirmos com nossa pasta `storage`. Veja o trecho abaixo:

```
1 Storage::disk('public')->delete($user->avatar);
```

Acima seleciono o disco `public`, para onde movemos os arquivos do avatar do usuário. Usando o método `delete` e informando o avatar do


```

1 if($request->hasFile('avatar')) {
2
3     $profileData['avatar'] =
$request->file('avatar')->store('avatars', 'public');
4
5 } else {
6     unset($profileData['avatar']);
7 }

```

Primeiramente verifico se a request possui o arquivo da imagem usando o método `hasFile` informando o nome do input, existindo, eu crio a chave `avatar` no array dentro de `$profileData` e uso os métodos `file` para recuperar a imagem enviada e o método `store` logo em seguida para upload, passando o nome da pasta `avatars` e o disco `public`, que resultará em nossa imagem salva dentro de `storage/app/public/avatars`.

O retorno deste upload eu mando para a chave `avatar` recém criada em `$profileData`, que ao atualizarmos o perfil receberá o nome da imagem bem como o nome da pasta `avatars` junto. Agora se formos ao nosso formulário, podemos testar o envio de uma foto para o perfil do usuário.

Pera, pera!!

Antes de testarmos, precisamos realizar mais uma melhoria no upload rsrsrs. Como vamos atualizar a foto do usuário precisamos remover a foto anterior, o arquivo no caso. Para isso precisamos usar o objeto `storage` para interagirmos com nossa pasta `storage`. Veja o trecho abaixo:

```
1 Storage::disk('public')->delete($user->avatar);
```

Acima seleciono o disco `public`, para onde movemos os arquivos do avatar do usuário. Usando o método `delete` e informando o avatar do

usuário removemos a foto da pasta `avatars`.*.

*Lembra que o nome da pasta é salva junto com o nome do arquivo lá no nosso banco, na tabela do profile.

Agora nosso trecho fica desta maneira:

```

1 if($request->hasFile('avatar')) {
2
3     Storage::disk('public')->delete($user->avatar);
4
5     $profileData['avatar'] = $request->file('avatar')->store('av
atars', 'public');
6
7 } else {
8     unset($profileData['avatar']);
9 }

```

PS.: Não esqueça de importar a classe:

```
1 use Illuminate\Support\Facades\Storage;
```

Testando upload de foto do perfil

Para testarmos vamos exibir a foto do usuário logo ao lado do seu nome no menu superior. Onde exibimos o nome do usuário e o menu dropdown para os links sair e profile, adicione a tag imagem como vemos adicionada abaixo no mesmo trecho:

```

1 <a id="navbarDropdown" class="nav-link dropdown-
toggle" href="#" role="button" data-\
2 toggle="dropdown" aria-haspopup="true" aria-
expanded="false" v-pre>

```



```

3      {{auth()->user()->name}}
4
5      @if(auth()->user()->profile->avatar)
6
7          user()->name}}}" class="rounded-
circle" width="50">
9
10     @endif
11
12     <span class="caret"></span>
13 </a>

```

A tag img:

```

1 user()->name}}}" class="rounded-circle" width="50">

```

Perceba que pego a imagem do caminho storage dentro do public do projeto e não lá de storage e concateno com o valor vindo do banco para o usuário autenticado: `auth()->user()->profile->avatar`.

Agora como linkar o conteúdo de storage lá para o public do projeto, bem simples, o laravel têm um comando que cria este link simbólico. Execute em seu terminal o comando abaixo:

```
1 php artisan storage:link
```

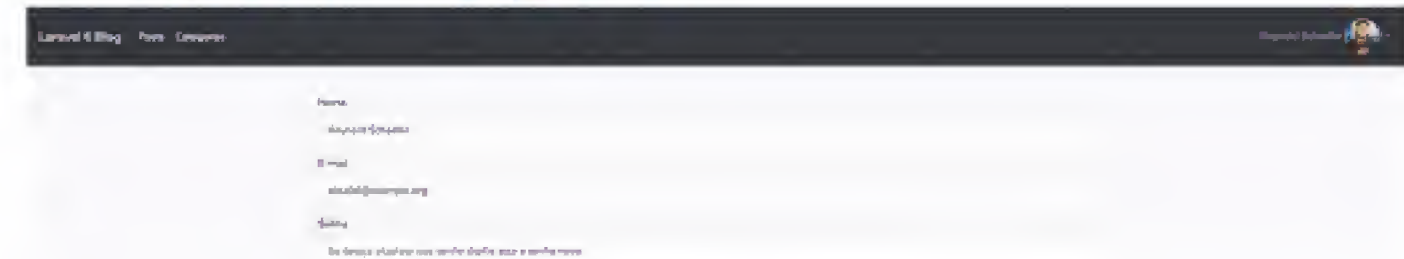
```

blog(master) x: php artisan storage:link
The [public/storage] directory has been linked.
blog(master) x:

```

Com isso o Laravel criará um link simbolico da pasta `storage/app/public` para a pasta `public/storage`. Sendo que `storage` é o link simbolico. A pasta `avatars` vai está lá como escolhemos no momento do upload no método `store`.

Veja o menu após a exibição da foto:



Vamos ao upload da foto de capa da postagem.

Upload de Capa Postagem

Primeiramente vamos criar a migration para adição da coluna `thumb` na tabela `posts`. Execute na raiz do seu projeto o comando abaixo:

```

1 php artisan make:migration
alter_table_posts_add_column_thumb --table=posts

```

```

blog(master): php artisan make:migration alter_table_posts_add_column_thumb --table=posts
Created Migration: 2019_10_23_135738_alter_table_posts_add_column_thumb

```

Segue na íntegra o conteúdo da migração:

```

1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6

```



```

7 class AlterTablePostsAddColumnThumb extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16
17         Schema::table('posts', function (Blueprint $table) {
18             $table->string('thumb')->nullable();
19         });
20
21     /**
22      * Reverse the migrations.
23      *
24      * @return void
25      */
26     public function down()
27     {
28
29         Schema::table('posts', function (Blueprint $table) {
30             $table->dropColumn('thumb');
31         });
32     }

```

Uma adição simples da coluna thumb tipo VARCHAR e permitindo valores nulos, o reverso é a remoção desta coluna.

Agora execute a migração na sua base, com o comando: `php artisan`

`migrate` e vamos prosseguir.

Obs.: Não esqueça de adicionar ela, a coluna thumb, lá no model Post, no array \$fillable.

Agora precisamos adicionar os inputs nos forms de criação e edição das postagens e não esqueça do atributo enctype na tag form. Veja os forms abaixo:

resources/views/posts/create.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="
5 {{route('posts.store')}}> method="post" enctype="multipart/form-data"
6
7     @csrf
8
9     <div class="form-group">
10         <label>Titulo</label>
11         <input type="text" name="title" class="form-control" value="{{old('title\
12 ')}}}">
13     </div>
14
15     <div class="form-group">
16         <label>Descrição</label>
17
18         <input type="text" name="description" class="form-control" value="{{old(\
19 'description')}}}">

```



```

20
21     <div class="form-group">
22         <label>Conteúdo</label>
23
24         <textarea name="content" id="" cols="30" rows="10" class="form-control">\
25             {{old('content')}}</textarea>
26     </div>
27
28     <div class="form-group">
29         <label>Slug</label>
30         <input type="text" name="slug" class="form-control" value="{{old('slug')}}">
31     </div>
32
33     <!-- Campo Tipo File -->
34
35     <div class="form-group">
36         <label>Foto de Capa</label>
37         <input type="file" name="thumb">
38     </div>
39
40     <!-- Campo Tipo File -->
41
42     <div class="form-group">
43         <label>Categorias</label>
44         <div class="row">
45             @foreach($categories as $c)
46                 <div class="col-2 checkbox">
47                     <label>
48
49                     <input type="checkbox" name="categories[]" value="{{ $c->name }}"

```

```

49         id}}"> {{ $c->name }}
50                 </label>
51             </div>
52         @endforeach
53     </div>
54 </div>
55
56     <div class="form-group">
57         <button class="btn btn-lg btn-success">Criar Postagem</button>
58     </div>
59 </form>
60 @endsection

```

resources/views/posts/edit.blade.php:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{ route('posts.update', ['post' =>
5         $post->id]) }}" method="post" enctype="multipart/form-data">
6
7         @csrf
8         @method("PUT")
9
10        <div class="form-group">
11            <label>Titulo</label>
12            <input type="text" name="title" class="form-control" value="{{ $post->title }}">
13        </div>
14
15        <div class="form-group">

```



```

17         <label>Descrição</label>
18
19         <input type="text" name="description" class="form-
20         control" value="{{ $post->description }}">
21     </div>
22
23     <div class="form-group">
24         <label>Conteúdo</label>
25
26         <textarea name="content" id="" cols="30" rows="10" class="fo
27         rm-control">
28             {{ $post->content }}</textarea>
29         </div>
30
31     <div class="form-group">
32         <label>Slug</label>
33
34         <input type="text" name="slug" class="form-
35         control" value="{{ $post->slug }}">
36     </div>
37
38     <!-- Campo Tipo File -->
39
40     <div class="form-group">
41         <label>Foto de Capa</label>
42
43         <input type="file" name="thumb">
44     </div>
45
46     <!-- Campo Tipo File -->
47
48     <div class="form-group">
49         <label>Categorias</label>

```

```

45     <div class="row">
46         @foreach($categories as $c)
47             <div class="col-2 checkbox">
48                 <label>
49
50                     <input type="checkbox" name="categories[]" value="{{ $c->
51                     id }}"
52
53                     @if($post->categories->contains($c)) checked @endif
54                     > {{ $c->name }}
55                 </label>
56             </div>
57         @endforeach
58     </div>
59
60     <div class="form-group">
61         <button class="btn btn-lg btn-
62         success">Atualizar Postagem</button>
63     </div>
64
65     </form>
66     <hr>
67     <form action="{{ route('posts.destroy', ['post' =>
68     $post->id]) }}" method="post">
69         @csrf
70         @method('DELETE')
71
72         <button type="submit" class="btn btn-lg btn-
73         danger">Remover Post</button>
74     </form>
75 @endsection

```

Os métodos update e store vão receber o trecho abaixo que já

conhecemos com excessão do update, o update terá a questão da imagem atual do storage:

Trecho a ser adicionado no método store do PostController:

```
1 if($request->hasFile('thumb')) {
2     $data['thumb'] =
$request->file('thumb')->store('thumbs', 'public');
3 } else {
4     unset($data['thumb']);
5 }
```

E no update:

```
1 if($request->hasFile('thumb')) {
2
3     //Remove a imagem atual
4     Storage::disk('public')->delete($post->thumb);
5
6
7     $data['thumb'] = $request->file('thumb')->store('thumbs', 'public');
8 } else {
9     unset($data['thumb']);
10 }
```

Agora basta testarmos o envio da foto da capa, tanto criando um post e depois na atualização.

No capítulo final onde criaremos o front do blog usaremos estas imagens de capa.

Conclusões

Trabalhar com upload de arquivos, em nosso caso específico fotos, é

bem simples no Laravel. O Laravel já traz todo o arcabouço pronto para isto, inclusive se precisarmos subir os arquivos no S3 da Amazon sem muito esforço.

Por enquanto temos um blog com muitas opções, mas ainda não estamos validando nenhum dos dados enviados para nossos controllers, no próximo capítulo iremos aplicar estas validações e entender como funcionam dentro do Laravel.

Até lá!

Validação

Olá tudo bem, estamos quase chegando na reta final do nosso ebook e neste capítulo iremos abordar sobre validações dentro do Laravel. Relembrando, sempre aplicando isso ao nosso projeto!

Podemos usar validação de duas maneiras em nossas aplicações. Uma delas é usando FormRequests e a outra é usando o objeto Validator e criando nossas validações customizadas e sob demanda.

Irei utilizar o FormRequest inicialmente, que cria uma camada extra de validação que não suja nossos controllers e fica totalmente isolada das nossas regras principais, é um forma de utilizar que se encaixa bem com todo o pensamento que estamos trazendo até aqui em nosso projeto.

Então o que é necessário para usar as validações em nossos módulos com Form Request? É isso que irei responder.

Vamos lá!

Form Request e Validações

Primeiramente vamos gerar nosso primeiro form request e logo em seguida iremos comentar sobre o códigos disponível nesta classe. Em seu terminal execute o comando abaixo na raiz do seu projeto:

```
1 php artisan make:request PostRequest
```

```
blog(master) X: php artisan make:request PostRequest
Request created successfully.
blog(master) X: █
```

Uma pasta será gerada dentro da pasta Http, a pasta Requests, e lá estará nosso PostRequest. Veja o conteúdo dele abaixo:

```
1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class PostRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      *
13      * @return bool
14      */
15     public function authorize()
16     {
17         return false;
18     }
19
20     /**
21      * Get the validation rules that apply to the request.
22      *
23      * @return array
24      */
25     public function rules()
26     {
27         return [
28             //
29         ];
30 }
```


O Form Request trará de cara dois métodos iniciais, o `authorize` e o `rules`. Vamos entender para que servem:

- **authorize**: Este método é disponível para verificar se determinado acesso está autorizado na sua aplicação, retornando `false` a requisição é automaticamente bloqueada na rota em que você utilizar este Form Request e retornando `true` a requisição passará normalmente caindo para as regras de validação em `rules`. No `authorize` você poderia verificar por exemplo se determinado usuário teria a permissão necessária para o acesso requisitado;
- **rules**: No método **rules** você define as regras para validação e assim que a requisição do formulário for enviada o Laravel usa estas regras e valida os dados antes mesmo de chegarem em seu controller e no método correspondente.

Podemos usar o Form Request para substituir o Request em nossos métodos que necessitam dele, isso trará a pitada extra de validação que será executada antes da requisição bater na execução do nosso método.

Vamo entender como montar as regras de nossa validação.

Montando Regras de Validação

A estrutura para as validações respeitam basicamente o nome dos campos dos inputs do seu formulário e o uso das validações disponíveis, para cada tipo de dado, disponibilizados pelo Laravel. O Laravel possui diversos validadores, como por exemplo, citando alguns e deixando referência para os outros:

- **required**: para campos obrigatórios;
- **email**: valida se um e-mail é válido;
- **unique**: garante que aquele valor é único no seu banco de dados e já testa se existe algum registro com aquele valor passado;
- **array**: valida se o valor é um array;
- **image**: valida se o valor é uma imagem válida: jpeg, png, bmp, gif,

svg, or webp.

- **confirm**: faz o match entre dois campos para verificar se os valores digitados são iguais. Excelente para confirmação de senha.

Existem diversos validadores e recomendo que você veja o que se enquadra melhor para a validação dentro da sua aplicação. Para visualizar as opções acesse: <https://laravel.com/docs/6.x/validation#available-validation-rules>.

Vamos montar nossas regras de validação para os campos do nosso formulário de criação e edição de postagens. Vamos lá.

Primeiramente irei colocar alguns campos como obrigatórios para termos um primeiro contato com as validações, veja o trecho do método `rules` do `PostRequest`:

```
1 /**
2  * Get the validation rules that apply to the request.
3  *
4  * @return array
5  */
6 public function rules()
7 {
8     return [
9         'title'          => 'required',
10        'description' => 'required',
11        'content'       => 'required',
12        'thumb'         => 'required',
13        'categories'    => 'required'
14    ];
15 }
```

Acima listei no array de retorno do método `rules` o nome dos nossos campos do formulário e defini para cada campo a validação para

campos obrigatórios. Podemos ainda utilizar mais validadores para cada um dos campos, e isso é possível quando usamos o pipe | e informamos outro validador para o campo escolhido. Se esse validador aceitar parâmetros nós informados por meio de um :, como por exemplo no validador abaixo.

Por exemplo posso colocar um tamanho mínimo ou máximo para nossa descrição(description), veja o método alterado:

```
1 /**
2  * Get the validation rules that apply to the request.
3  *
4  * @return array
5  */
6 public function rules()
7 {
8     return [
9         'title'      => 'required',
10        'description' => 'required|min:20',
11        'content'     => 'required',
12        'thumb'       => 'required',
13        'categories'  => 'required'
14    ];
15 }
```

Acima digo que nosso campo de descrição além de ser obrigatório, também têm um valor mínimo a ser digitado e esse valor mínimo, informado depois do : é 20 caracteres.

Posso ainda, adicionar mais um validador para nosso campo da thumb do post, para a validação de imagens:

```
1 /**
2  * Get the validation rules that apply to the request.
```

```
3  *
4  * @return array
5  */
6 public function rules()
7 {
8     return [
9         'title'      => 'required',
10        'description' => 'required|min:20',
11        'content'     => 'required',
12        'thumb'       => 'required|image',
13        'categories'  => 'required'
14    ];
15 }
```

Agora, como utilizar o Form Request e também exibir estas validação em nossas views?

Vamos por parte, vamos continuando.

Usando o Form Request

É um processo bem simples, só precisamos trocar a referência nos métodos, trocando o Request pelo PostRequest nos métodos store e update. Que estão assim:

store:

```
1 ...
2 public function store(Request $request)
3 {
4 ...
```

update:

```
1 ...
```



```
2 public function update(Post $post, Request $request)
3 {
4 ...
```

Ficarão assim:

store:

```
1 ...
2
3 public function store(PostRequest $request)
4 {
5
6 ...
```

update:

```
1 ...
2 public function update(Post $post, PostRequest $request)
3 {
4 ...
```

Simples, simples assim. Não precisamos alterar mais nada nos controllers, isso se dá por que o Form Request estende do Request por isso não precisamos alterar nada e ainda teremos acesso aos métodos para manipulação do dados vindos da requisição.

Obs.: Como não vamos utilizar regras no método authorize do PostRequest ao invés de retornar false retorne true.

Veja o PostRequest na íntegra:

```
1 <?php
2
3 namespace App\Http\Requests;
4
```

```
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class PostRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      * @return bool
13      */
14     public function authorize()
15     {
16         return true;
17     }
18
19     /**
20      * Get the validation rules that apply to the request.
21      *
22      * @return array
23      */
24     public function rules()
25     {
26         return [
27             'title'          => 'required',
28             'description' => 'required|min:20',
29             'content'       => 'required',
30             'thumb'         => 'required|image',
31             'categories'    => 'required'
32         ];
33     }
34 }
```

Agora precisamos testar essas validações e exibir pro usuário as

mensagens dos validadores retornados pelo Laravel, para cada uma das validações especificadas. Agora sim, vamos exibir as validações e suas mensagens em nossas telas.

Exibindo validações nas Views Blade

O Laravel disponibiliza uma variável para ser acessada em nossas views chamada de `$errors` que receberá um array com as validações que não passaram durante o envio dos dados do formulário. Mas, temos outras possibilidades dentro do Laravel para tratarmos e exibirmos os erros e ainda prover uma forma de exibição para cada campo do formulário de forma mais direta e simplificada.

Por meio da diretiva `@error` podemos tratar estas exibições de forma bem simples, vamos entender como ela funciona.

Como sabemos, aplicamos as validações com base nos nomes dos campos de nossos formulários pois são esses os identificadores das informações enviadas em nossa requisição. Para recuperarmos erros específicos de cada campo podemos usar a diretiva `@error` como abaixo:

```
1 @error('title')
2
<h1>Existe um erro de validação para o input do titulo do pos
t</h1>
3 @enderror
```

A diretiva retornará true caso exista um erro de validação para o campo informado. Assim você pode exibir mensagens de controle para este campo, ou mesmo pegar a mensagem lançada pelo proprio Laravel. Para isso basta exibir o valor da variável `$message` criada pela diretiva e que contém a mensagem real do erro aplicado pela validação.

Veja:

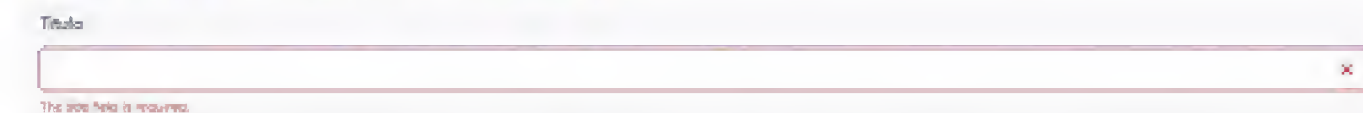
```
1 @error('title')
2     <h1>{{ $message }}</h1>
3 @enderror
```

Acima, se existir erros de validação para o campo `title` a mensagem do erro será exibida dentro do `h1`.

Agora vamos entender como usar esta diretiva em nossos formulários com o Bootstrap.

Exibindo validações nos formulário com Bootstrap

O Twitter Bootstrap possui uma classe chamada de `is-invalid` para exibição de erro em determinado campo do formulário. Essa classe adiciona uma borda vermelha ao seu input e você ainda pode usar uma área abaixo do input dentro de um `p`(parágrafo) por exemplo, para exibição da mensagem customizada, essa mensagem recebe a classe `css` chamada de `invalid-feedback`.



Veja acima o resultado da combinação das classes `is-invalid` e da área da mensagem do erro que recebe a classe `invalid-feedback`.

O código usado para a exibição acima, você pode ver abaixo:

```
1 <div class="form-group">
2     <label>Titulo</label>
3
4     <input type="text" name="title" class="form-control is-
invalid" value="{{old('ti\
5 tle')}}">
```



```

6
7     <p class="invalid-feedback">The title field is
required.</p>
8 </div>

```

Agora só precisamos combinar a diretiva @error para exibir as classes de validação quando as validações não passarem. Pegando o trecho acima vamos adicionar a diretiva e depois mostro todo o formulário alterado com esse pensamento. Veja o trecho mencionado acima com a diretiva @error aplicada abaixo:

```

1 <div class="form-group">
2     <label>Titulo</label>
3     <input type="text" name="title" class="form-control
@error('title') is-invalid @\
4 enderror" value="{{old('title')}}">
5
6     @error('title')
7         <p class="invalid-feedback">{{$message}}</p>
8     @enderror
9 </div>

```

Caso existam erros de validação existentes para o campo title, a diretiva irá retornar true, e com isso podemos adicionar a classe is-invalid e ainda exibir o bloco com a mensagem da validação. Conforme já comentamos dentro da diretiva é possível utilizar a variável \$message para recuperar o erro real da validação, conforme vamos acima.

Agora é só replicarmos este pensamento para cada um dos inputs. Veja o formulário completo abaixo:

```

1 @extends('layouts.app')
2
3 @section('content')

```

```

4     <form action="{{route('posts.store')}}" method="post" enctype="multipart/form-data">
5
6         @csrf
7
8         <div class="form-group">
9             <label>Titulo</label>
10            <input type="text" name="title" class="form-control
@error('title') is-invalid
12 nvalid @enderror" value="{{old('title')}}">
13
14            @error('title')
15                <p class="invalid-feedback">
{{$message}}</p>
16            @enderror
17        </div>
18
19        <div class="form-group">
20            <label>Descrição</label>
21
22            <input type="text" name="description" class="form-control
@error('description') is-invalid @enderror" value="{{old('description')}}">
23
24            @error('description')
25                <p class="invalid-feedback">
{{$message}}</p>
26            @enderror
27        </div>
28
29        <div class="form-group">

```



```

30         <label>Conteúdo</label>
31
32         <textarea name="content" id="" cols="30" rows="10" class="form-control @\
33 error('content') is-invalid @enderror">{{old('content')}}
34 </textarea>
35         @error('content')
36         <p class="invalid-feedback">
37 {{ $message }}</p>
38         @enderror
39     </div>
40
41     <div class="form-group">
42         <label>Slug</label>
43         <input type="text" name="slug" class="form-
44 control" value="{{old('slug')}}"
45 }}">
46     </div>
47
48     <div class="form-group">
49         <label>Foto de Capa</label>
50         <input type="file" name="thumb" class="form-
51 control @error('thumb') is-i\
52 nvalid @enderror">
53         @error('thumb')
54         <p class="invalid-feedback">
55 {{ $message }}</p>
56         @enderror
57     </div>
58
59     <div class="form-group">
60         <label>Categorias</label>
61         <div class="form-group">
62             <div class="form-group">
63                 <input type="checkbox" class="form-check-input"
64                 @error('categories') is-invalid
65                 @enderror" name="categories[]" value="{{ $c->id }}"
66                 <label class="form-check-label">
67                     {{ $c->name }}
68                 </label>
69             </div>
70             @endforeach
71         </div>
72     </div>
73 </form>
74 @endsection

```

```

56         @foreach($categories as $c)
57             <div class="col-2 custom-control
58 custom-checkbox">
59                 <input type="checkbox" class="custom-control-input
60 @error('c\
61 ategories') is-invalid
62 @enderror" name="categories[]" value="{{ $c->id }}"
63                 <label class="custom-control-
64 label">
65                     {{ $c->name }}
66                 </label>
67             </div>
68         @endforeach
69     </div>
70
71     <div class="form-group">
72         <button class="btn btn-lg btn-
73 success">Criar Postagem</button>
74     </div>
75 </form>
76 @endsection

```

O de edição também seguirá o mesmo pensamento, veja o formulário de edição alterado abaixo:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{ route('posts.update', ['post' =>
5 $post->id]) }}" method="post" e\
6 nctype="multipart/form-data">

```



```

6      @csrf
7      @method("PUT")
8
9      <div class="form-group">
10         <label>Titulo</label>
11         <input type="text" name="title" class="form-
12 control @error('title') is-i\
13 nvalid @enderror" value="{{ $post->title }}">
14         @error('title')
15         <p class="invalid-feedback">
16 {{ $message }}</p>
17         @enderror
18     </div>
19
20     <div class="form-group">
21         <label>Descrição</label>
22         <input type="text" name="description" class="form-control
23 @error('descr\
24 iption') is-invalid @enderror" value="
25 {{ $post->description }}">
26         @error('description')
27         <p class="invalid-feedback">
28 {{ $message }}</p>
29         @enderror
30     </div>
31
32     <div class="form-group">
33         <label>Conteúdo</label>
34         <textarea name="content" id="" cols="30" rows="10" class="fo
35 rm-control \

```

```

36 @error('content') is-invalid @enderror">{{ $post->content }}
37 </textarea>
38
39         @error('content')
40         <p class="invalid-feedback">
41 {{ $message }}</p>
42         @enderror
43     </div>
44
45     <div class="form-group">
46         <label>Slug</label>
47         <input type="text" name="slug" class="form-
48 control" value="{{ $post->slug\
49 }}">
50     </div>
51
52     <div class="form-group">
53         <label>Foto de Capa</label>
54         <input type="file" name="thumb" class="form-
55 control @error('thumb') is-\
56 invalid @enderror">
57         @error('thumb')
58         <p class="invalid-feedback">
59 {{ $message }}</p>
60         @enderror
61     </div>
62
63     <div class="form-group">
64         <label>Categorias</label>
65         <div class="form-group">
66             @foreach($categories as $c)
67                 <div class="col-2 custom-control
68 custom-checkbox">
69

```



```

    <input type="checkbox" class="custom-control-input
@error('c\
58 ategories') is-invalid
@enderror" name="categories[]" value="{{ $c->id }}"
59
@if($post->categories->contains($c)) checked @endif>
60         <label class="custom-control-
label">
61             {{ $c->name }}
62         </label>
63
64     </div>
65 @endforeach
66 </div>
67 </div>
68
69 <div class="form-group">
70     <button class="btn btn-lg btn-
success">Atualizar Postagem</button>
71 </div>
72
73 </form>
74 <hr>
75 <form action="{{ route('posts.destroy', ['post' =>
$post->id]) }}" method="post">
76     @csrf
77     @method('DELETE')
78     <button type="submit" class="btn btn-lg btn-
danger">Remover Post</button>
79 </form>
80 @endsection

```

OBS.: Eu percebi que não adicionamos a classe form-control para o input file do thumb do post. Adicione essa classe no input file dos forms

de edição e criação de uma nova postagem.

OBS 2: Fiz algumas alterações no input checkbox que rege a adição das categorias deste post. O código acima já está alterado mas para mostrar, o input checkbox que estava assim:

create:

```

1 <div class="form-group">
2     <label>Categorias</label>
3     <div class="row">
4         @foreach($categories as $c)
5             <div class="col-2 checkbox">
6                 <label>
7
8                 <input type="checkbox" name="categories[]" value="
{{ $c->id }}"> {\
9 {{ $c->name }}
10
11                 </label>
12
13             </div>
14         @endforeach
15     </div>
16 </div>

```

update:

```

1 <div class="form-group">
2     <label>Categorias</label>
3     <div class="row">
4         @foreach($categories as $c)
5             <div class="col-2 checkbox">
6                 <label>

```



```

7
<input type="checkbox" name="categories[]" value="
{{$c->id}}"
8
@if($post->categories->contains($c)) checked @endif
9
    > {{$c->name}}
10
    </label>
11
    </div>
12
    @endforeach
13
</div>
14 </div>

```

Está assim nos códigos das telas agora:

create:

```

1 <div class="form-group">
2     <label>Categorias</label>
3     <div class="form-group">
4         @foreach($categories as $c)
5             <div class="col-2 custom-control custom-
checkbox">
6                 <input type="checkbox" class="custom-
control-input @error('categorie\
7 s') is-invalid @enderror" name="categories[]" value="
{{$c->id}}">
8                 <label class="custom-control-label">
9                     {{$c->name}}
10                </label>
11
12            </div>
13        @endforeach
14    </div>
15 </div>

```

edit:

```

1 <div class="form-group">
2     <label>Categorias</label>
3     <div class="form-group">
4         @foreach($categories as $c)
5             <div class="col-2 custom-control custom-
checkbox">
6                 <input type="checkbox" class="custom-
control-input @error('categorie\
7 s') is-invalid @enderror" name="categories[]" value="
{{$c->id}}">
8                 @if($post->categories->contains($c)) checked @endif>
9                     <label class="custom-control-label">
10                         {{$c->name}}
11                     </label>
12
13            </div>
14        @endforeach
15    </div>
16 </div>

```

Fiz algumas alterações de classe e na estrutura para adicionar as classes de validação e adequar com Bootstrap 4.

Validação em Categorias

Como é repeteco vamos nas instruções diretamente!

Primeiramente vamos gerar nosso CategoryRequest para criarmos nossas regras de validação. Execute a geração em seu terminal com o comando abaixo:

```
1 php artisan make:request CategoryRequest
```


Veja abaixo o conteúdo do CategoryRequest completo e com as regras de validação:

```

1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class CategoryRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      * @return bool
13      */
14     public function authorize()
15     {
16         return true;
17     }
18
19     /**
20      * Get the validation rules that apply to the request.
21      *
22      * @return array
23      */
24     public function rules()
25     {
26         return [
27             'name' => 'required'
28         ];
29     }
30 }

```

Troque os requests do store e do update do CategoryController assim como fizemos no PostController. E por fim veja os formulários alterados da área de categorias:

create.blade.php

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="
5 {{route('categories.store')}}>" method="post">
6         @csrf
7
8         <div class="form-group">
9             <label>Nome</label>
10             <input type="text" name="name" class="form-
11 control @error('name') is-inv\
12 alid @enderror" value="{{old('name')}}">
13             @error('name')
14                 <p class="invalid-feedback">
15                     {{ $message }}
16                 </p>
17             @enderror
18         </div>
19
20         <div class="form-group">
21             <label>Descrição</label>
22
23             <input type="text" name="description" class="form-
24 control" value="{{old(\

```



```

25         <div class="form-group">
26             <label>Slug</label>
27             <input type="text" name="slug" class="form-
control" value="{{old('slug')\
28 }}">
29         </div>
30
31         <button class="btn btn-lg btn-
success">Criar Categoria</button>
32     </form>
33 @endsection

```

edit.blade.php

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="{{route('categories.update', ['category'
=> $category->id])}}" met\
5     hod="post">
6
7         @csrf
8         @method("PUT")
9
10        <div class="form-group">
11            <label>Nome</label>
12            <input type="text" name="name" class="form-
control @error('name') is-inv\
13 alid @enderror" value="{{ $category->name }}">
14
15            @error('name')
16                <p class="invalid-feedback">
17                    {{ $message }}
18                </p>

```

```

19        @enderror
20    </div>
21
22    <div class="form-group">
23        <label>Descrição</label>
24
25        <input type="text" name="description" class="form-
control" value="{{ $cat\
26 egory->description }}">
27    </div>
28
29    <div class="form-group">
30        <label>Slug</label>
31        <input type="text" name="slug" class="form-
control" value="{{ $category->\
32 slug }}">
33    </div>
34
35    <button class="btn btn-lg btn-
success">Atualizar Categoria</button>
36 </form>
37 @endsection

```

Validamos aqui apenas o nome da categoria que de fato é obrigatório, até para gerarmos os slugs automáticos no próximo capítulo.

Vamos as validações no perfil do usuário. Vamos lá!

Validações Perfil Usuário

Vamos gerar nosso Form Request para o perfil do usuário, chamarei ele de UserProfileRequest. Execute o comando abaixo no seu terminal para geração do nosso Form Request:

```
1 php artisan make:request UserProfileRequest
```


Veja o conteúdo dele na íntegra:

```

1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class UserProfileRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      *
13      * @return bool
14      */
15     public function authorize()
16     {
17         return true;
18     }
19
20     /**
21      * Get the validation rules that apply to the request.
22      *
23      * @return array
24      */
25     public function rules()
26     {
27         return [
28             'user.name'      => 'required',
29             'user.email'     => 'required|email',
30             'profile.avatar' => 'image'
31         ];
32     }

```

32 }

Aqui entra um ponto diferente em relação aos nomes dos input referenciados pela validação e com base no atributo name de cada um deles. Como separamos os inputs em duas casas, uma para os dados do user (da tabela usuários) e um pro perfil (da tabela de perfis) precisamos referenciar o nome do campo respeitando o formato chamado dentro do atributo name. Por exemplo:

```

1 <input type="text" name="user[name]" class="form-control
  @error('user.name') is-inva\
2 lid @enderror" value="{{ $user->name }}">

```

O nome do input acima é user[name] quando formos referenciar o nome deste input para validação precisamos seguir o seguinte formato: user.name. Assim o Laravel saberá qual campo ele estará validando, essa chamada se dá por que receberemos um array dentro da chave user, assim como na chave profile também e cada um tendo seus campos em questão.

Agora, basta chamarmos o form request lá no método update do UserProfileController, como vemos abaixo:

```

1 ...
2 public function update(UserProfileRequest $request)
3 ...

```

E nossa view alterada segue abaixo:

```

1 @extends('layouts.app')
2
3 @section('content')
4     <form action="
5 {{route('profile.update')}}" method="post" enctype="multipart
6 /form\

```



```

5 -data">
6
7     @csrf
8
9     <div class="form-group">
10         <label>Nome</label>
11
12         <input type="text" name="user[name]" class="form-control
13         @error('user.name') is-invalid @enderror" value="{{ $user->name }}"
14         @error('user.name')
15             <div class="invalid-feedback">
16                 {{ $message }}
17             </div>
18         @enderror
19     </div>
20
21     <div class="form-group">
22         <label>E-mail</label>
23
24         <input type="text" name="user[email]" class="form-control
25         @error('user.email') is-invalid @enderror" value="{{ $user->email }}"
26         @error('user.email')
27             <div class="invalid-feedback">
28                 {{ $message }}
29             </div>
30         @enderror
31     </div>
32
33     <div class="form-group">
34         <label>Senha</label>

```

```

<input type="password" name="user[password]" class="form-
control" placeholder="Senha" value="">
35 <div class="form-group">
36     <label>Senha atual</label>
37
38     <div class="form-group">
39         <label>Sobre</label>
40
41         <textarea name="profile[about]" id="" cols="30" rows="10" class="form-control">{{ $user->profile->about }}</textarea>
42     </div>
43
44     <div class="form-group">
45         <label>Avatar</label>
46         <input type="file" name="avatar" class="form-control @error('avatar') is-
47         invalid @enderror">
48
49         @error('user.email')
50             <div class="invalid-feedback">
51                 {{ $message }}
52             </div>
53         @enderror
54     </div>
55
56
57
58     <div class="form-group">
59         <label>Facebook</label>
60
61         <input type="url" name="profile[facebook_link]" class="form-control" value="">

```



```

61 ue="{{ $user->profile->facebook_link }}">
62     </div>
63     <div class="form-group">
64         <label>Instagram</label>
65         <input type="url" name="profile[instagram_link]" class="form-
-control" value=
66         "{{ $user->profile->instagram_link }}">
67     </div>
68     <div class="form-group">
69         <label>Site</label>
70         <input type="url" name="profile[site_link]" class="form-
-control" value="\
71         "{{ $user->profile->site_link }}">
72     </div>
73
74     <div class="form-group">
75         <button class="btn btn-lg btn-
success">Atualizar Meu Perfil</button>
76     </div>
77 </form>
78 @endsection

```

Mensagens de erro

Dentro do form request podemos ainda traduzir as mensagens de erro, simplesmente sobrescrevendo o método `messages` que está no pai do nosso form request em questão. Por exemplo, vamos traduzir as mensagens de validação lá do `PostRequest`, logo após o método `rules` adicione o método abaixo:

```

1 public function messages()
2 {
3     return [
4         'required' => 'Este campo é obrigatório',
5         'min'      => 'Sua descrição deve ter pelo menos
:min caracteres',
6         'image'    => 'Imagem inválida'
7     ];
8 }

```

O método `messages` retornará um array, referenciando o nome dos validadores como índice de cada linha e para cada linha, referentes ao validador, o valor será a mensagem que você quer que seja exibida. Agora sempre que a validação ocorrer as mensagens escolhidas irão aparecer.

Note que consigo acessar, na chave `min` o valor digitado para a validação de quantidade mínima de caracteres, por meio da notação `:min`.

Para saber mais sobre as possibilidades com respeito as mensagens de validações recomendo a documentação: <https://laravel.com/docs/6.x/validation#working-with-error-messages>.

Veja o `PostRequest` na íntegra:

```

1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class PostRequest extends FormRequest
8 {
9     /**

```



```

10     * Determine if the user is authorized to make this
    request.
11     *
12     * @return bool
13     */
14     public function authorize()
15     {
16         return true;
17     }
18
19     /**
20     * Get the validation rules that apply to the request.
21     *
22     * @return array
23     */
24     public function rules()
25     {
26         return [
27             'title'      => 'required',
28             'description' => 'required|min:20',
29             'content'    => 'required',
30             'thumb'      => 'required|image',
31             'categories' => 'required'
32         ];
33     }
34
35     public function messages()
36     {
37         return [
38             'required' => 'Este campo é obrigatório',
39             'min'      => 'Sua descrição deve ter pelo
menos :min caracteres',
40             'image'    => 'Imagem inválida'

```

```

41         ];
42     }
43 }

```

Veja também o CategoryRequest e o UserProfileRequest já com o método messages adicionado:

CategoryRequest:

```

1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class CategoryRequest extends FormRequest
8 {
9     /**
10     * Determine if the user is authorized to make this
    request.
11     *
12     * @return bool
13     */
14     public function authorize()
15     {
16         return true;
17     }
18
19     /**
20     * Get the validation rules that apply to the request.
21     *
22     * @return array
23     */
24     public function rules()

```



```

25     {
26         return [
27             'name' => 'required'
28         ];
29     }
30
31     public function messages()
32     {
33         return [
34             'required' => 'Este campo é obrigatório'
35         ];
36     }
37 }

```

UserProfileRequest:

```

1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class UserProfileRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      *
13      * @return bool
14      */
15     public function authorize()
16     {
17         return true;
18     }
19 }

```

```

18
19     /**
20      * Get the validation rules that apply to the request.
21      *
22      * @return array
23      */
24     public function rules()
25     {
26         return [
27             'user.name' => 'required',
28             'user.email' => 'required|email'
29         ];
30     }
31
32     public function messages()
33     {
34         return [
35             'required' => 'Este campo é obrigatório',
36             'email' => 'E-mail digitado é inválido'
37         ];
38     }
39 }

```

##Validações Customizadas

Para praticarmos e conhecermos as validações customizadas, vamos implementar uma validação de senha pro perfil do usuário.

Porque validação customizada cabe aqui? Como os form requests realizam a validação antes mesmo de chegar no nosso método não iríamos conseguir validar a senha neste momento pois ela só é alterada quando o usuário digita algo no input de senha.

Meu principal intuito aqui é mostrar pra você como criar validação

customizada dada a sua necessidade então vou validar aqui somente o tamanho da senha que não pode ser menor que 8 caracteres. Agora, como fazemos isso?

O Laravel possui a classe Validator que nos permite realizar as validações conhecidas dentro do Form Request mas sob demanda ou de forma customizada. Primeiro passo, importe a classe Validator do namespace:

```
1 use Illuminate\Support\Facades\Validator;
```

A classe Validator possui o método make que nos permite criarmos nossas regras de validação, o primeiro parâmetro deste método são os dados que você deseja validar, neste caso, nossos dados vindos na requisição; O segundo parâmetro é o array com as validações que já conhecemos, temos ainda o terceiro parâmetro que compete as mensagens e é onde podemos, neste cenário, traduzir as mensagens para os validadores utilizados e por fim o método make ainda possui um quarto parâmetro para atributos customizados.

Os três primeiros são os que nos interessam aqui. Primeiramente vamos chamar a classe Validator lá dentro do if da senha no UserProfileController, dentro do método update, e acessar o método make como vemos abaixo:

```
1 $validator = Validator::make(
2     $request->all(),
3     [
4         'user.password' =>
5         ['min:8']
6     ],
7     [
8         'min' => 'Senha deve ter
9         pelo menos :min caracteres!'
```

```
8     ]
9 );
10
11 if($validator->fails()) {
12     return redirect()->back()->withErrors($validator);
13 }
```

Primeiramente passo todos os dados vindos na request por meio do método all, segundo, defino meus validadores. Aqui só valido o tamanho da senha, no caso, o mínimo aceito são 8 caracteres e por fim o terceiro parâmetro, onde apenas jogo uma mensagem traduzida para ser exibida durante a validação.

Após isso, eu pego o objeto Validator com nossas regras definidas e verifico se a validação falhou usando o método fails() na condicional. Se falhou, quer dizer que o usuário digitou menos que 8 caracteres então dentro do if eu redireciono ele de volta com os erros pegos dentro do nosso validador customizado. Por esta razão chamo após o método back o método withErrors passando todo o validador contido na variável \$validator. Com isso os erros serão enviados de volta para a tela de perfil e serão exibidos lá para o campo password (senha).

Ah, não esqueça de adicionar a exibição do erro para o campo de senha lá na view do profile (resources/views/profile/index.blade.php), veja o campo abaixo:

```
1 <div class="form-group">
2     <label>Senha</label>
3
4     <input type="password" name="user[password]" class="form-
5     control @error('user.pa\
6     ssword') is-invalid @enderror" placeholder="Se deseja
7     atualizar sua senha digite aq\
8     ui a senha nova...">
```



```

6     @error('user.password')
7     <div class="invalid-feedback">
8         {{$message}}
9     </div>
10    @enderror
11 </div>

```

Com a classe Validator fica fácil criarmos validações customizadas e sob demanda como foi o caso que vimos aqui. Para entender melhor onde adicionar a validação, abaixo deixo o método update completo após a alteração lá no UserProfileController. Veja:

```

1 public function update(UserProfileRequest $request)
2 {
3     $userData = $request->get('user');
4     $profileData = $request->get('profile');
5
6     try{
7
8         if($userData['password']) {
9             //Aqui começa a validação customizada...
10
11 $validator = Validator::make($request->all(),
12     [
13         'user.password' => ['min:8']
14     ],
15     ['min' => 'Senha deve ter pelo menos :min
16     caracteres!']);
17
18     if($validator->fails()) {
19         return redirect()->back()->withErrors($validator);
20     }
21 }

```

```

20
21 $userData['password'] = bcrypt($userData['password']);
22     } else {
23         unset($userData['password']);
24     }
25
26     $user = auth()->user();
27
28     if($request->hasFile('avatar')) {
29         Storage::disk('public')->delete($user->avatar);
30
31         $profileData['avatar'] = $request->file('avatar')->store('av
32     atars', 'pub\
33     lic');
34     } else {
35         unset($profileData['avatar']);
36     }
37
38     $user->update($userData);
39
40     $user->profile()->update($profileData);
41
42     flash('Perfil atualizado com sucesso!')->success();
43     return redirect()->route('profile.index');
44 } catch(\Exception $e) {
45
46     $message = 'Erro ao remover categoria!';
47
48     if(env('APP_DEBUG')) {
49         $message = $e->getMessage();
50     }
51 }

```



```

52     flash($message)->warning();
53     return redirect()->back();
54
55 }
56 }

```

Bom, então é isso!

Conclusões

Neste capítulo nós conhecermos as principais formas de trabalho com validações de dados dentro do Laravel, uma usando uma camada extra, os Forms Requests e a outra usando o proprio Validator para criar validações cutomizadas ou sob demanda para nossos dados.

O Laravel possui diversos validadores e as possibilidades são imensas além da simplicidade e facilidade na utilização destes recursos e foi o que vimos aqui. Para concluirmos nosso livro, no próximo capítulo vamos criar a interface pública do nosso blog para deixarmos nosso blog mais redondo e completo!

Então, até o próximo capítulo!

Criando Front do nosso Blog

Olá, tudo bem? Espero que sim!

Chegamos ao nosso último capítulo e aqui vamos criar o front do nosso blog. Com a navegação de posts, navegação de posts por categorias além de criarmos também uma área de comentários para cada postagem de nosso blog.

E não podemos esquecer, ainda vamos dinamizar a criação dos slugs para as postagens e também para as categorias!

Então vamos lá!

Front do Blog

Primeiramente vamos gerar os dois controllers para nosso front:

- HomeController: Responsável pela home e pela single da postagem;
- CategoryController: Responsável pela listagem de posts por categorias.

Irei gerar estes controllers dentro da pasta Site, que também será seus namespaces. Para gerarmos, como já conhecemos, execute na raiz do projeto os comandos abaixo, um após o outro:

```
1 php artisan make:controller Site/HomeController
```

```
1 php artisan make:controller Site/CategoryController
```

Agora vamos focar no HomeController e realizar a navegação home e single das postagens, vamos lá!

Listagem de Posts e Single

Iremos listar as postagens na home paginando estas postagens, além de ordená-las de forma decrescentes e ainda criar a tela da postagem ou single como costumamos chamar.

Veja o código completo do HomeController que possui dois métodos:

- index: Tela inicial com todos os posts;
- single: Tela única da postagem ou single.

Veja abaixo:

```
1 <?php
2
3 namespace App\Http\Controllers\Site;
4
5 use App\Post;
6 use Illuminate\Http\Request;
7 use App\Http\Controllers\Controller;
8
9 class HomeController extends Controller
10 {
11     /**
12      * @var Post
13      */
14     private $post;
15
16     public function __construct(Post $post)
17     {
18         $this->post = $post;
19     }
20
21     public function index()
22     {
23
24         $posts = $this->post->orderBy('id', 'DESC')->paginate(15);
```

```
25         return view('site.posts.index', compact('posts'));
26     }
27
28     public function single($slug)
29     {
30         $post = $this->post->whereSlug($slug)->first();
31
32         return view('site.posts.single', compact('post'));
33     }
34 }
```

Aqui vale alguns comentários, primeiramente no método index. Antes de chamarmos o paginate do Eloquent, utilizamos o método orderBy para ordenarmos as postagens de forma decrescente trazendo assim as mais recentes primeiro. Reforço a simplicidade que é realizar esta operação dentro do Eloquent uma vez que o nome dos métodos são muito intuitivos.

No método single uso uma abordagem interessante que é chamando o where junto com o campo no nome do método, como está lá: whereSlug. O Laravel, via Eloquent vai entender que ele tem que buscar o post onde o slug seja igual o slug recebido na url via parâmetro.

Isso simplifica muito a escrita de condicionais em queries, e logo após o método whereSlug uso o método first para pegar o resultado desta query e o primeiro e único resultado.

Abaixo segue as views na íntegra. As views estão dentro da pasta de views, na pasta site/posts(crie a pasta site e a pasta posts dentro da pasta site):

index.blade.php

```
1 @extends('layouts.site')
2
```



```

3
4 @section('content')
5 <div class="row">
6     <div class="col-8">
7         <div class="col-12">
8             <h2>Postagens</h2>
9             <hr>
10        </div>
11        @foreach($posts as $post)
12            <div class="col-12">
13                @if($post->thumb)
14                    
17                @else
18                    
21                @endif
22                <h3>{{ $post->title }}</h3>
23                <p>
24                    {{ $post->description }}
25                </p>
26                <a href="{{route('site.single', ['slug' =>
27 $post->slug])}}>">Leia mai\
28 s...</a>
29                <hr>
30            </div>
31        @endforeach
32    <div class="col-12">
33        {{ $posts->links() }}
34    </div>
35</div>

```

```

33    <div class="col-4">
34        <div class="col-12">
35            <h2>Sidebar</h2>
36            <hr>
37        </div>
38    </div>
39</div>
40</div>
41@endsection

```

Na view acima, `index.blade.php`, temos uma pequena condição para exibição da thumb da postagem. Caso a postagem não tenha uma thumb na base nós exibimos uma foto padrão.

Você pode baixar esta foto padrão e adicionar dentro da pasta `img` dentro da pasta `public` do projeto. Se a pasta `img` não existir crie ela e jogue a imagem `no-photo.jpg` dentro. Acesse a imagem em <https://codeexperts.com.br/books/no-photo.jpg>.

Continuando...

single.blade.php

```

1 @extends('layouts.site')
2
3
4 @section('content')
5     <div class="row">
6         <div class="col-8">
7             <div class="col-12">
8                 <h2>{{ $post->title }}</h2>
9                 <hr>
10            </div>
11
12            <div class="col-12">

```



```

13         @if($post->thumb)
14             
16         @else
17             
19         @endif
20         <p>
21             {!! $post->content !!}
22         </p>
23     </div>
24 </div>
25 </div>
26 <div class="col-4">
27     <div class="col-12">
28         <h2>Sidebar</h2>
29         <hr>
30     </div>
31 </div>
32 </div>
33 </div>
34 @endsection

```

Perceba a linha do {!! \$post->content !!}, aqui uso um print diferenciado. Por padrão o print {{}}, do blade, escapa as entidades html para prevenção de XSS, entretanto este conteúdo da postagem pode vir com parágrafos e determinadas formatações que precisamos interpretar por isso ao invés do {{}} usei o {!! !!}.

Perceba que nas views temos um template associado a elas, template esse a qual elas estendem. Neste caso crie dentro da pasta layouts na pasta de views o arquivo site.blade.php com o conteúdo abaixo:

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no,
initial-scale=1.0, maximum-\
7 scale=1.0, minimum-scale=1.0">
8     <meta http-equiv="X-UA-Compatible" content="ie=edge">
9     <title>Blog Code Experts</title>
10    <link rel="stylesheet" href="{{asset('css/app.css')}}>">
11    <style>
12        .navbar {
13            margin-bottom: 40px;
14        }
15    </style>
16 </head>
17 <body>
18    <nav class="navbar navbar-expand-lg navbar-light bg-
light">
19        <a class="navbar-brand" href="/">Laravel 6 Blog</a>
20        <button class="navbar-toggler" type="button" data-
toggle="collapse" data-tar\
21 get="#navbarNavDropdown" aria-
controls="navbarNavDropdown" aria-expanded="false" ari\
22 a-label="Toggle navigation">
23            <span class="navbar-toggler-icon"></span>
24        </button>
25        <div class="collapse navbar-

```



```

collapse" id="navbarNavDropdown">
26         <ul class="navbar-nav">
27             <li class="nav-item active">
28                 <a class="nav-link" href="{{
route('site.index') }}">Home</a>
29             </li>
30         </ul>
31     </div>
32
33     @auth
34         <ul class="navbar-nav ml-auto">
35             <li class="nav-item dropdown">
36                 <a id="navbarDropdown" class="nav-
link dropdown-toggle" href\
37 ="#" role="button" data-toggle="dropdown" aria-
haspopup="true" aria-expanded="false"\
38 v-pre>
39                     {{auth()->user()->name}}
40
41                     user()->name}}" class="rounded-circle" width="50">
43
44                     <span class="caret"></span>
45                 </a>
46
47                 <div class="dropdown-menu dropdown-
menu-right" aria-labelledby\
48 by="navbarDropdown">
49                     <a class="dropdown-
item" href="{{ route('logout') }}">
50
51

```

```

onclick="event.preventDefault();
52
document.getElementById('logout-form\
53 ').submit();">
54
Sair
55
</a>
56
<form id="logout-
57 form" action="{{ route('logout') }}" me\
58 thod="POST" style="display: none;">
59
@csrf
60
</form>
61
<a class="dropdown-
62 item" href="{{ route('profile.index')\
63 }}">
64
Profile
65
</a>
66
</div>
67
</li>
68
</ul>
69
@endauth
70
</nav>
71
<div class="container">
72
@include("flash::message")
73
@yield('content')
74
</div>
75
<script src="{{asset('js/app.js')}}"></script>
76
77 </body>
78 </html>

```

E por último adicione as rotas, no web.php, para acesso das telas

criadas acima:

```
1 Route::namespace('Site')->name('site.')->group(function(){
2     Route::get('/', 'HomeController@index')->name('index');
3     Route::get('/post/{slug}', 'HomeController@single')->name('single');
4 });
```

Obs.: Comente ou remova os trechos do arquivo web.php mostrados abaixo para evitar conflitos de mesmo nome da rota:

Rota para a tela principal vinda na geração do projeto

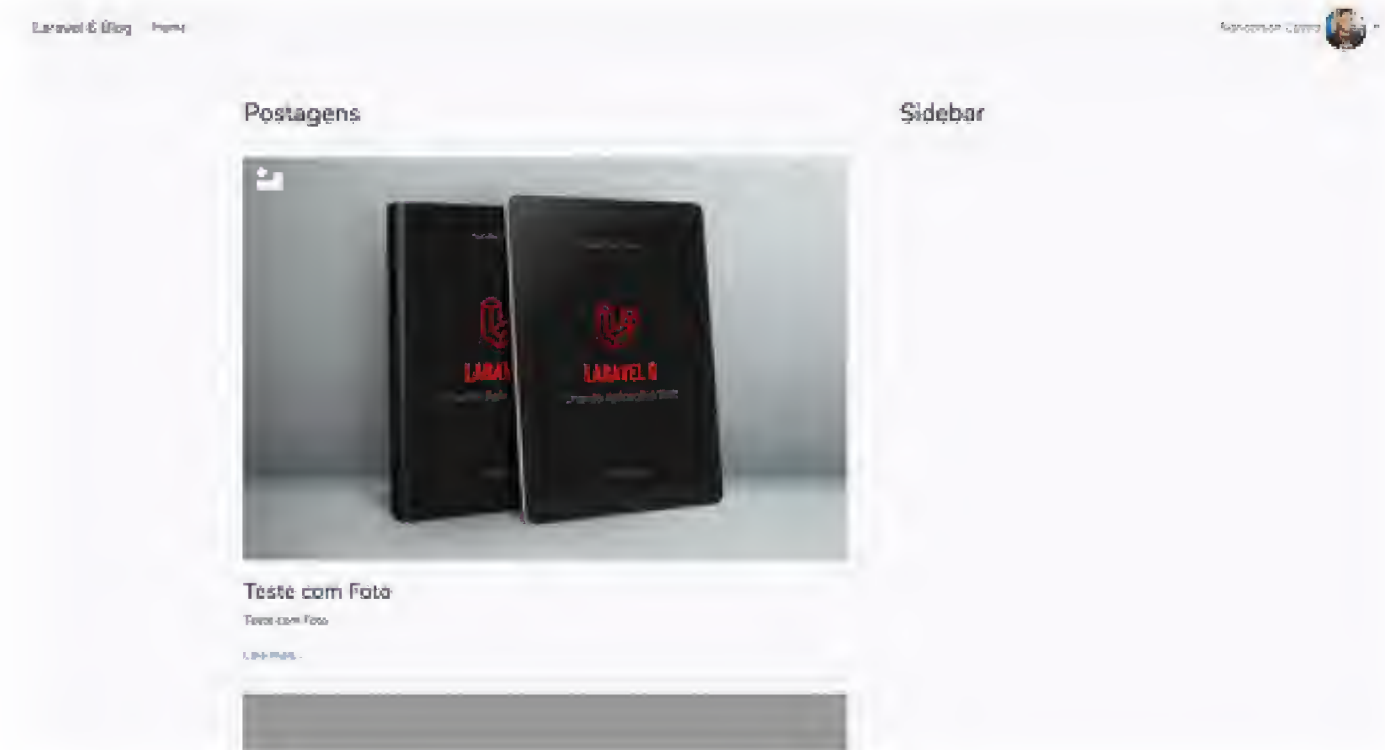
```
1 Route::get('/', function () {
2     return view('welcome');
3 });
```

E também nossa rota teste para parâmetros dinâmicos

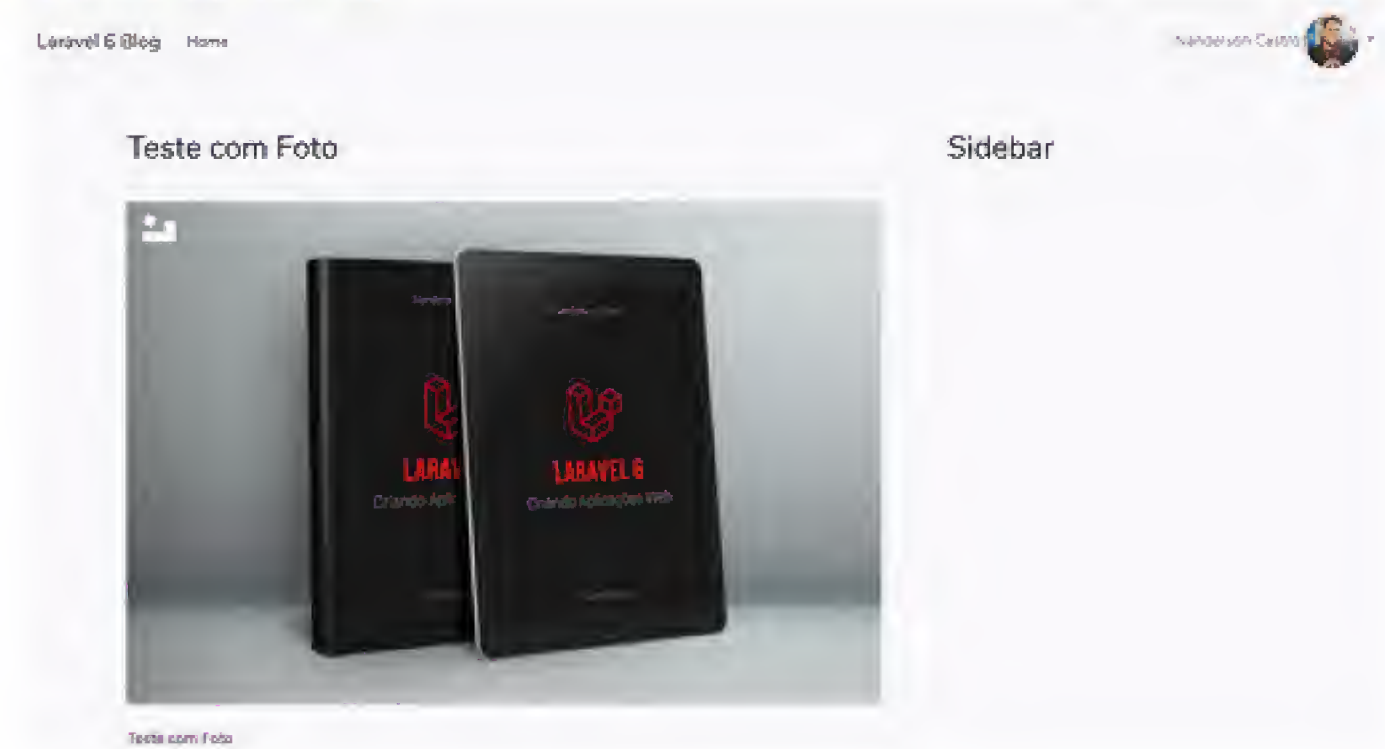
```
1 Route::get('/post/{slug}', function($slug) {
2     return $slug;
3 });
```

Feita estas modificações, veja como ficou as telas:

Home do Blog



Tela da Postagem



Obs.: Se você estiver logado o menu do canto superior direito irá aparecer como visto nas imagens acima.

Agora vamos incrementar mais a tela da postagem de nosso blog adicionando a possibilidade de inclusão de comentários para cada postagem.

Criando Comentários

Primeiramente vamos iniciar os pontos necessários para nosso sistema de comentários e mais a frente só linkamos as coisas dentro da página single das postagens.

Para isso gere o model Comment e sua migration com o comando abaixo:

```
1 php artisan make:model Comment -m
```

Abaixo deixo o conteúdo da migration criada, que no momento que criei recebeu o nome: 2019_12_15_202745_create_comments_table.php.

Veja o código da migration:

```
1 <?php
2
3 use Illuminate\Support\Facades\Schema;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Database\Migrations\Migration;
6
7 class CreateCommentsTable extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
```

```
14     public function up()
15     {
16         Schema::create('comments', function (Blueprint $table) {
17             $table->bigIncrements('id');
18             $table->unsignedBigInteger('post_id');
19             $table->text('comment');
20             $table->boolean('status');
21             $table->timestamps();
22
23             $table->foreign('post_id')->references('id')->on('posts');
24         });
25     }
26
27     /**
28      * Reverse the migrations.
29      *
30      * @return void
31      */
32     public function down()
33     {
34         Schema::dropIfExists('comments');
35     }
36 }
```

Feita a criação da migration, execute a mesma em seu projeto com o comando:

```
1 php artisan migrate
```

Agora vamos associar os models Post e Comment criando a relação entre eles via models. A relação aqui será de Um para Muitos / Muitos para Um onde Um Post pode ter Vários Comentários e um Comentário

pertence a apenas um Post.

Associando Comentários e Posts

Em **Post.php** adicione o seguinte método dentro de seu model:

```
1 public function comments()
2 {
3     return $this->hasMany(Comment::class);
4 }
```

Veja o **Post.php** completo pós adição:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Post extends Model
8 {
9     protected $fillable = [
10         'title',
11         'description',
12         'content',
13         'slug',
14         'is_active',
15         'user_id',
16         'thumb'
17     ];
18
19     public function user()
20     {
21         return $this->belongsTo(User::class);
22     }
```

```
23
24     public function categories()
25     {
26
27         return $this->belongsToMany(Category::class, 'posts_categories');
28     }
29
30     public function comments()
31     {
32         return $this->hasMany(Comment::class);
33     }
```

Agora adicione o trecho abaixo dentro do **Comment.php**:

```
1 public function post()
2 {
3     return $this->belongsTo(Post::class);
4 }
```

Veja o conteúdo do **Comment.php** na íntegra pós alteração:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Comment extends Model
8 {
9     protected $fillable = ['comment', 'status'];
10
11     public function post()
12     {
```



```

13         return $this->belongsTo(Post::class);
14     }
15 }

```

Feita estas associações mapeando a relação entre os dois models acima estamos aptos a criar de fato a adição de comentários para as postagens de nosso blog.

Salvando Comentários

Aqui temos poucos campos e uma ligação com o post a qual vai pertencer o comentário. Mais conteúdo que já conhecemos.

Agora vamos criar o controller de comentários para podermos receber a requisição post para a criação do comentário enviado pelo usuário.

Gere o controller com o comando abaixo:

```
1 php artisan make:controller Site/CommentController
```

Veja nosso controller na íntegra com o método saveComment na íntegra:

```

1 <?php
2
3 namespace App\Http\Controllers\Site;
4
5 use App\Http\Requests\CommentRequest;
6 use App\Http\Controllers\Controller;
7 use App\Post;
8
9 class CommentController extends Controller
10 {
11     public function saveComment(CommentRequest $request)
12     {
13         try {
14             $comment = $request->get('comment');

```

```

15
16             $post = Post::find($request->get('post_id'));
17             $post->comments()->create([
18                 'comment' => $comment,
19                 'status' => true
20             ]);
21
22             flash('Comentário criado com
sucesso!')->success();
23
24             return redirect()->route('site.single', ['slug' => $post->sl
ug]);
25
26         } catch (\Exception $e) {
27             $message = 'Erro ao criar comentário!';
28
29             if(env('APP_DEBUG')) {
30                 $message = $e->getMessage();
31             }
32
33             flash($message)->warning();
34             return redirect()->back();
35         }
36 }

```

Do nosso formulário enviarei dois campos, o campo `post_id` e o campo `comment`. Criamos o comentário via ligação onde buscamos o post pela referência vinda do formulário de comentários na chave `post_id`.

O comentário de fato vem na chave `comment` onde passo no array para o método `create`, via ligação com `post`. Com isso teremos o comentário criado já recebendo a referência da postagem.

Perceba que habilito o status sempre como `true` mas você pode colocar

como false e liberar no painel uma tela para o usuário gerenciar estes comentários e liberar apenas os comentários sensatos.

Adicione também validação para o campo do comentário, então gere o form request:

```
1 php artisan make:request CommentRequest
```

E adicione o código abaixo:

```
1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6
7 class CommentRequest extends FormRequest
8 {
9     /**
10      * Determine if the user is authorized to make this
11      * request.
12      *
13      * @return bool
14      */
15     public function authorize()
16     {
17         return true;
18     }
19
20     /**
21      * Get the validation rules that apply to the request.
22      *
23      * @return array
24      */
25     public function rules()
```

```
25     {
26         return [
27             'comment' => 'required'
28         ];
29     }
30 }
```

Este request já está linkado em nosso controller CommentController. Vamos prosseguir!

Para exibição dos comentários bem como de seu formulário de criação separei um arquivo que vamos incluir dentro da single da postagem. Então crie dentro da pasta site uma pasta includes e dentro desta pasta crie um arquivo comments.blade.php com o conteúdo abaixo:

```
1 <div class="col-12">
2     <hr>
3     <h3>Comentários</h3>
4     <hr>
5     <form action="
6     {{route('site.single.comment')}}" method="post">
7         @csrf
8         <input type="hidden" name="post_id" value="
9         {{$post->id}}">
10        <div class="form-group">
11            <label>Seu Comentário</label>
12            <textarea name="comment" class="form-control
13            @error('comment') is-invalid">
14            {{old('comment')}}
15            </textarea>
16
17            @error('comment')
18                <div class="invalid-feedback">
19                    {{$message}}
20                </div>
21            </div>
22        </form>
23    </div>
```



```

16         </div>
17         @enderror
18     </div>
19     <button type="submit" class="btn btn-lg btn-
success">Enviar Comentário</butt\
20 on>
21 </form>
22 </div>
23
24 @if($post->comments->count())
25 <div class="col-12">
26     <hr>
27     <h3>Comentários</h3>
28     <hr>
29
30     @foreach($post->comments()->orderBy('id', 'DESC')->get() as
$comment)
31         <div class="col-12">
32             <small>Comentário enviado em {{date('d/m/Y H:i:s', strtotime
($comment->creat\
33 ed_at))}}</small>
34             <p>{{ $comment->comment}}</p>
35         </div>
36     @endforeach
37 </div>
38 @endif

```

Acima temos o form de criação do comentário com o input hidden para o campo post_id e um textarea para o comentário em si. Um detalhe importante para analisarmos com calma é o trecho da listagem dos comentários, destaco ele abaixo para comentarmos:

```

1 @if($post->comments->count())
2 <div class="col-12">
3     <hr>
4     <h3>Comentários</h3>
5     <hr>
6
7     @foreach($post->comments()->orderBy('id', 'DESC')->get() as
$comment)
8         <div class="col-12">
9             <small>Comentário enviado em {{date('d/m/Y H:i:s', strtotime
($comment->creat\
10 ed_at))}}</small>
11             <p>{{ $comment->comment}}</p>
12         </div>
13     @endforeach
14 </div>
15 @endif

```

A exibição da listagem dos comentários é condicionada a existência de comentários para a postagem acessada. Para isso uso o count vindo da collection de comentários por meio da ligação:

```
1 $post->comments->count()
```

Outro ponto é a chamada do método orderBy via ligação para ordenar os comentários pelos mais recentes, dentro dos comentários da postagem acessada:

```
1 $post->comments()->orderBy('id', 'DESC')->get()
```

Este são dois pontos importantes a salientarmos neste ponto da listagem de comentários. Agora, vamos incluir este arquivo comments.blade.php dentro do arquivo single.blade.php.

Posicionei a linha abaixo, logo após o col-12 do conteúdo da postagem:

```
1 @include('site.includes.comments')
```

Veja a **single.blade.php** completa pós inclusão do arquivo de comentários:

```
1 @extends('layouts.site')
2
3
4 @section('content')
5     <div class="row">
6         <div class="col-8">
7             <div class="col-12">
8                 <h2>{{ $post->title }}</h2>
9                 <hr>
10            </div>
11
12            <div class="col-12">
13                @if($post->thumb)
14                    
17                @else
18                    
21                @endif
22                <p>
23                    {!! $post->content !!}
24                </p>
25            </div>
26            @include('site.includes.comments')
```

```
26     </div>
27     <div class="col-4">
28         <div class="col-12">
29             <h2>Sidebar</h2>
30             <hr>
31         </div>
32     </div>
33 </div>
34 </div>
35 @endsection
```

Um último passo é expor a rota para envio do comentário/criação. Dentro do grupo de rotas que criamos para a home e a single adicione o trecho abaixo:

```
1 Route::post('/post/comment',
2 'CommentController@saveComment')->name('single.comment\');
```

Uma rota post apontando para o método saveComment do CommentController.

Com isso você pode testar o envio de comentários para uma postagem. Veja como ficou a tela da postagem agora:



Teste com Foto

Comentários

Crie Comentário

Enviar Comentário

Comentários

Comentário enviado em 10/13/2018 22:02:48

Outro comentário teste

Comentário enviado em 10/13/2018 22:01:32

Enviando comentário para post

Postagens por Categorias

Para começarmos vamos iniciar o método index para listagem das postagens por categoria. Quando iniciamos o capítulo já realizamos a geração do nosso controller para este trabalho, o CategoryController.

Sem mais delongas veja o método index, do CategoryController, a ser adicionado:

```
1 public function index($slug)
2 {
3     $category = $this->category->whereSlug($slug)->first();
4     $posts = $category->posts()->paginate(15);
5
6     return view('site.category', compact('category',
7     'posts'));
```

Primeiramente pegamos a categoria pelo slug dela e logo após buscamos as postagens desta categoria, paginei estes posts via ligação para termos nossa paginação também nesta listagem de postagens por categoria.

Agora precisamos enviar os dados para nossa view, nossa view será muito parecida com a view de postagens da home o que muda é que vamos exibir um título com o nome da categoria. Veja a view completa category.blade.php criada dentro de resources/views/site/:

category.blade.php

```
1 @extends('layouts.site')
2
3
4 @section('content')
5     <div class="row">
6         <div class="col-8">
7             <div class="col-12">
8                 <h2>Categoria: {{ $category->name }}</h2>
9                 <hr>
10            </div>
11            @foreach($posts as $post)
12                <div class="col-12">
13                    @if($post->thumb)
14                        
17                    @else
18                        
21                    @endif
22                    <h3>{{ $post->title }}</h3>
```



```

21         <p>
22             {{$post->description}}
23         </p>
24         <a href="{{route('site.single', ['slug'
=> $post->slug])}}">Leia\
25     mais...</a>
26         <hr>
27     </div>
28     @empty
29         <div class="alert alert-
warning">Sem posts para esta categoria!</div>
30     @endforelse
31     <div class="col-12">
32         {{$posts->links()}}
33     </div>
34 </div>
35 <div class="col-4">
36     <div class="col-12">
37         <h2>Sidebar</h2>
38         <hr>
39     </div>
40 </div>
41 </div>
42 </div>
43 @endsection

```

Com nossa view criada precisamos expor nossa rota para acesso desta tela. Dentro do grupo de rotas para o site/front adicione a rota abaixo:

```

1 Route::get('/category
/{slug}', 'CategoryController@index')->name('category');

```

O trecho completo com as rotas do site/front estão abaixo:

```

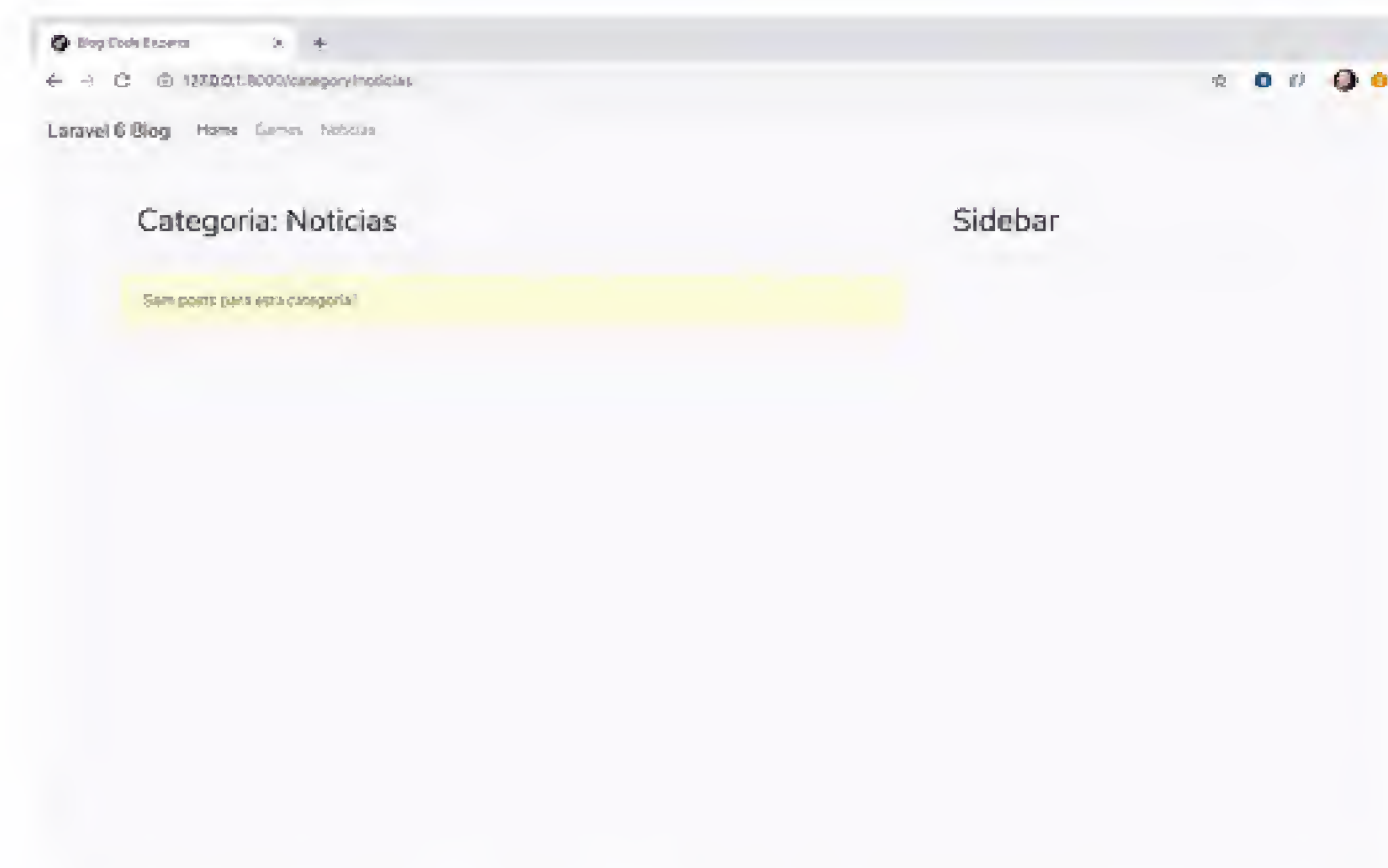
1 Route::namespace('Site')->name('site.')->group(function(){
2     Route::get('/', 'HomeController@index')->name('index');
3     Route::get('/post
/{slug}', 'HomeController@single')->name('single');
4
5     Route::post('/post
/comment', 'CommentController@saveComment')->name('single.com
ment\
6 ');
7
8     Route::get('/category
/{slug}', 'CategoryController@index')->name('category');
9 });

```

Agora se acessarmos o link `http://127.0.0.1:8000/category/[slug]` onde [slug] seja um slug de uma categoria existente, conseguiremos acessar os posts por categoria. Por exemplo, tenho aqui as categorias noticias e games. Então, acessando `http://127.0.0.1:8000/category/games` tenho o resultado abaixo:



E quando a categoria não possui postagens temos a tela abaixo:



Agora precisamos expor os links das categorias em nosso menu como você já pode ver na imagem acima, então vamos a isso.

Compartilhando Categorias entre as Views

Exibir as categorias em cada view é um ponto que precisamos realizar, entretanto, não podemos ficar repetindo essas buscas em todos os controllers responsáveis pelas telas do front.

Para isso vamos compartilhar as categorias com todas as views de forma a sempre existir a variável com as categorias existentes em nosso blog.

Vamos adicionar esta chamada ao provedor principal de uma aplicação Laravel, o `AppServiceProvider` que você pode encontrar dentro da pasta `app/Providers`. Nesta classe podemos encontrar dois métodos:

- register: serve para registro de serviços para sua aplicação;
- boot: serve para configurações de inicialização dos serviços de sua aplicação.

Vamos adicionar nosso dado comum para todas as views no método boot do AppServiceProvider. Dentro do método boot adicione a seguinte linha abaixo:

```
1
view()->share(['categories' => \App\Category::all('name', 'slug')]);
```

Aqui chamamos a função view sem parâmetros, onde recebemos um View/Factory como resultado, partir deste View/Factory podemos usar o método share que nos permite compartilhar parâmetros entre todas as nossas views.

Onde seto uma chave categories que recebe uma busca por todas as categorias do nosso blog. De cada categoria pegarei apenas o nome e o slug de cada uma delas, estas duas informações são mais que necessárias para montarmos nossos links.

Veja o AppServiceProvider completo agora:

```
1 <?php
2
3 namespace App\Providers;
4
5 use Illuminate\Support\ServiceProvider;
6
7 class AppServiceProvider extends ServiceProvider
8 {
9     /**
10      * Register any application services.
11      *
```

```
12      * @return void
13      */
14     public function register()
15     {
16     }
17
18     /**
19      * Bootstrap any application services.
20      *
21      * @return void
22      */
23     public function boot()
24     {
25         view()->share(['categories' => \App\Category::all('name', 'slug')]);
26     }
27 }
```

Exibindo categorias no menu

Agora temos em todas as views a possibilidade de acessar as categorias por meio da variável \$categories, enviado pelo método share visto anteriormente.

Agora é só realizarmos um loop e montarmos os links lá dentro do nosso layout site.blade.php. Logo após do li do link da home adicione o trecho abaixo:

```
1 @foreach($categories as $category)
2 <li class="nav-item">
3     <a class="nav-link" href="{{ route('site.category',
4         ['slug' => $category->slug])\"
```



```

4 }}">{{ $category->name }}</a>
5 </li>
6 @endforeach

```

Desta forma veremos as categorias cadastradas em nosso blog compondo o menu principal do nosso front. Para conferência veja o código do layout, `site.blade.php`, completo para análise:

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no,
7         initial-scale=1.0, maximum-\
8         scale=1.0, minimum-scale=1.0">
9     <meta http-equiv="X-UA-Compatible" content="ie=edge">
10    <title>Blog Code Experts</title>
11    <link rel="stylesheet" href="{{asset('css/app.css')}}">
12    <style>
13        .navbar {
14            margin-bottom: 40px;
15        }
16    </style>
17 </head>
18 <body>
19     <nav class="navbar navbar-expand-lg navbar-light bg-
20     light">
21         <a class="navbar-brand" href="/">Laravel 6 Blog</a>
22         <button class="navbar-toggler" type="button" data-
23         toggle="collapse" data-tar\
24         get="#navbarNavDropdown" aria-
25         controls="navbarNavDropdown" aria-expanded="false" ari\

```

```

22 a-label="Toggle navigation">
23         <span class="navbar-toggler-icon"></span>
24     </button>
25     <div class="collapse navbar-
26     collapse" id="navbarNavDropdown">
27         <ul class="navbar-nav">
28             <li class="nav-item active">
29                 <a class="nav-link" href="{{
30                 route('site.index') }}">Home</a>
31             </li>
32             @foreach($categories as $category)
33                 <li class="nav-item">
34                     <a class="nav-link" href="{{
35                     route('site.category', ['slug' \
36                     => $category->slug]) }}">{{ $category->name }}</a>
37                 </li>
38             @endforeach
39         </ul>
40     </div>
41
42     @auth
43         <ul class="navbar-nav ml-auto">
44             <li class="nav-item dropdown">
45                 <a id="navbarDropdown" class="nav-
46                 link dropdown-toggle" href\
47                 ="#" role="button" data-toggle="dropdown" aria-
48                 haspopup="true" aria-expanded="false"\
49                 v-pre>
50                     {{auth()->user()->name}}
51
52                     <img src="{{asset('storage/' .
53                     auth()->user()->profile->\

```



```

49 avatar}}}" alt="Foto de
  {{auth()->user()->name}}" class="rounded-circle" width="50">
50
51         <span class="caret"></span>
52     </a>
53
54     <div class="dropdown-menu dropdown-
menu-right" aria-labelledby\
55 by="navbarDropdown">
56         <a class="dropdown-
item" href="{{ route('logout') }}"
57
58         onclick="event.preventDefault();
59         document.getElementById('logout-form\
59 ').submit();">
60             Sair
61         </a>
62
63         <form id="logout-
form" action="{{ route('logout') }}" me\
64 thod="POST" style="display: none;">
65             @csrf
66         </form>
67
68         <a class="dropdown-
item" href="{{ route('profile.index')\"
69 }}">
70             Profile
71         </a>
72     </div>
73 </li>
74 </ul>

```

```

75         @endauth
76     </nav>
77     <div class="container">
78         @include("flash::message")
79         @yield('content')
80     </div>
81
82     <script src="{{asset('js/app.js')}}"></script>
83 </body>
84 </html>

```

Com isso chegamos ao final da nossa aplicação!

Antes de concluirmos precisamos dinamizar os slugs de nossas postagens e das nossas categorias.

Dinamizando Geração de Slugs Post & Category

Para dinamizar a geração de slugs em meus projetos Laravel eu costumo usar um pacote chamado de `spatie/laravel-sluggable`. Então na raiz do seu projeto execute o comando abaixo:

```
1 composer require spatie/laravel-sluggable
```



```

blog(master): composer require spatie/laravel-sluggable

Using version ^2.2 for spatie/laravel-sluggable
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing spatie/laravel-sluggable (2.2.1): Loading from cache
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: facade/ignition
Discovered Package: fideloper/proxy
Discovered Package: laracasts/flash
Discovered Package: laravel/tinker
Discovered Package: laravel/ui
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
blog(master) x: █

```

O interessante deste pacote é que ele evita o conflito de slugs e existindo o slug em questão ele incrementa evitando, assim, este conflito. Após a instalação precisamos dizer em nossos models, qual a coluna o pacote deve lê para gerar o slug e onde ele deve salvar este slug.

Veja o trecho que adicionaremos em nossos models para este trabalho:

```

1 public function getSlugOptions() : SlugOptions
2 {
3     return SlugOptions::create()
4         ->generateSlugsFrom('name')
5         ->saveSlugsTo('slug');
6 }

```

Este método vêm da trait HasSlug, que adicionaremos, e diz exatamente

que o nosso slug deve ser gerado a partir da coluna name (método generateSlugsFrom) e deve ser salvo na coluna slug (método saveSlugsTo).

Os dois imports vêm do namespace Spatie\Sluggable, de pegaremos a trait HasSlug e o SlugOptions. Por exemplo:

```
1 use Spatie\Sluggable\{SlugOptions, HasSlug};
```

Isso realmente é bem simples, veja nossos dois models completos com as aplicações mencionados acima.

Post.php:

```

1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6 use Spatie\Sluggable\{SlugOptions, HasSlug};
7
8 class Post extends Model
9 {
10     use HasSlug;
11
12     protected $fillable = [
13         'title',
14         'description',
15         'content',
16         'slug',
17         'is_active',
18         'user_id',
19         'thumb'
20     ];
21

```



```

22     public function getSlugOptions() : SlugOptions
23     {
24         return SlugOptions::create()
25             ->generateSlugsFrom('title')
26             ->saveSlugsTo('slug');
27     }
28
29     public function user()
30     {
31         return $this->belongsTo(User::class);
32     }
33
34     public function categories()
35     {
36
37     return $this->belongsToMany(Category::class, 'posts_categorie
38 s');
39
40     }
41
42     public function comments()
43     {
44         return $this->hasMany(Comment::class);
45     }
46 }

```

Category.php:

```

1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
6 use Spatie\Sluggable\{SlugOptions, HasSlug};
7
8 class Category extends Model

```

```

9 {
10     use HasSlug;
11
12     protected $fillable = ['name', 'description', 'slug'];
13
14     public function getSlugOptions() : SlugOptions
15     {
16         return SlugOptions::create()
17             ->generateSlugsFrom('name')
18             ->saveSlugsTo('slug');
19     }
20
21     public function posts()
22     {
23
24     return $this->belongsToMany(Post::class, 'posts_categories');
25 }

```

Agora é testar! E você já pode remover os campos slug dos formulários de criação e edição de posts e categorias, pois a geração já está automatizada. Agora, mesmo que tenha categoria com o mesmo nome ou postagem com o mesmo título, os slugs gerados não gerarão duplicidades e serão sempre únicos.

Com isso, concluímos de fato nosso projeto! Vamos as conclusões!

Conclusões

Chegamos ao ponto final da nossa aplicação planejada para este livro, um blog com painel gerenciável. Este capítulo foi bem mais fluído onde usamos dos conhecimentos já adquiridos no decorrer do livro para compormos as telas da nossa aplicação na visão pública.

Utilizamos ainda do compartilhamento de parâmetros entre as views

para entregarmos nossas categorias com todas as nossas views e utilizarmos para montagem dos links em nossas telas para a exibição das postagens por categorias.

Com isso espero que todo o conhecimento adquirido aqui possa ter te ajudado imensamente para prosseguir para novos horizontes aplicando este conhecimento em seus projetos futuros, e claro, não parar a busca dos conhecimentos para melhor entrega de projetos usando o Laravel Framework.

Mais uma vez obrigado! E meus sinceros desejo de sucesso! Felicidades!

Continue em contato conosco

Continue em contato conosco e se especializando com a Code Experts, temos diversos cursos na área de desenvolvimento web com PHP e frameworks focando totalmente em prática como você pode acompanhar aqui neste livro.

Se você gosta de aprender colocando a mão na massa acesse já codeexperts.com.br.

Mais uma vez obrigado e meu sinceros de desejo de sucesso sempre!

